

**ОБЪЕКТНО-
ОРИЕНТИРОВАННОЕ
ПРОЕКТИРОВАНИЕ**
с примерами применения

OBJECT ORIENTED DESIGN

WITH APPLICATIONS

Grady Booch

Rational

The Behjamen/Cummings Publishing Company, Inc.

Redwood City, California * Fort Collins, Colorado * Menlo Park, California *
Redding, Massachusetts * New York * Don Mills, Ontario * Wokingham, U.K.
Amsterdam * Bonn * Sydney * Singapore * Tokyo * Madrid * San Juan

Г. БУЧ

**ОБЪЕКТНО-
ОРИЕНТИРОВАННОЕ
ПРОЕКТИРОВАНИЕ**

с примерами применения

Перевод с английского
А. А. ИВАНОВА, А. В. КАРНАУХОВА
и М. В. ЩЕЛКИНА

Под редакцией
А. Н. АРТАМОШКИНА

Совместное издание
Фирмы "Диалектика" г. Киев
и АО "И. В. К." г. Москва

ББК 32.973
Б94
УДК 681.3.06

Буч Г.

Б94
Объектно-ориентированное проектирование с примерами применения: Пер. с англ. — М.: Конкорд, 1992.— 519 с., ил.

ISBN 5-87737-002-2

Книга американского специалиста представляет собой первое полное изложение объектно-ориентированной методологии: анализ, проектирование, программирование. В книге рассмотрены фундаментальные вопросы объектного подхода, практические аспекты конструирования программных систем. Отдельный раздел книги посвящен примерам использования различных объектно-ориентированных языков программирования в реальных системах. В книге содержится обширная библиография по предметной области. Книга рассчитана на профессиональных программистов, руководителей больших программных проектов и студентов, будущая профессия которых связана с разработкой сложных программных систем.

Б 2404010000-002 — без объявл.
87737-92

ББК 32.973

Научное издание

Гради Буч

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ
С ПРИМЕРАМИ ПРИМЕНЕНИЯ

Подписано в печать 07.09.92. Формат 70х100/16. Бумага офсетная. Гарнитура "Таймс-Роман". Печать офсетная. Усл. печ. л. 42,57. Уч.-изд. л. 46,32. Тираж 10000 экз.

Оригинал-макет подготовлен в издательстве ТОО "Конкорд". 129090, Москва, ул. Щепкина, 22, пом. 19.

Полиграфические работы выполнены АО «И.В.К.». 129090, Москва, ул. Щепкина, 22.

ISBN 0-8053-0091-0(англ.)

© 1991 by The Benjamin/Cummings Publishing Company Inc.

ISBN 5-87737-002-2 (русс.)

© Перевод на русский язык и предисловие к русскому изданию ТОО «Конкорд», 1992 г.

Отпечатано с компьютерного набора на комбинате печати издательства "Пресса України"

252047, Киев-47, проспект Победы, 50. Зак. 0212151.

Предисловие редактора перевода

Книга вышла в январе 1991 г. В январском номере американского журнала *Journal of object-oriented programming (JOOP Jan 1991, vol. 3, No 5)* в своей рецензии на книгу редактор журнала д-р Ричард С. Винер подчеркнул: «Буч затронул очень сложную тему в этой нужной книге... Использование объектно-ориентированного проектирования пока распространено относительно нешироко. Книга Буча позволяет значительно продвинуться в этой предметной области. И новички и опытные практики получат большую пользу, прочитав эту книгу».

Гради Буч — основатель и Директор американской фирмы *Object-Oriented Products at Rational*, образованной в 1980 г. Г-н Буч был одним из первых в США, кто стал использовать метод объектно-ориентированного проектирования и применять различные объектные и объектно-ориентированные языки программирования в своих разработках. Г-ном Бучем написано две книги: «*Software Engineering with Ada*» (1986) и «*Object Oriented Design with Applications*». Г-н Буч является членом редколлегии американских журналов *Object Magazine*, *Journal of Object-Oriented Programming*, *HotLine of Object-Oriented Technology*. Кроме этого г-н Буч выступает с лекциями и докладами на ежегодно проводимых международных семинарах, конференциях и курсах обучения по объектно-ориентированному подходу, которые проводятся в Европе и США.

Ценность книги в том, что она является первым полным изложением Объектно-Ориентированной Методологии и ее компонент, а также превосходным учебником, практическим руководством по объектно-ориентированному проектированию — направлению, которому в настоящее время уделяется большое внимание в США и Европе. Книга безусловно интересна профессионалам — руководителям больших проектов, системотехникам, программистам. Она будет интересна и в случае, если Вы вообще никогда не слышали ранее о существовании объектно-ориентированного подхода, но интересуетесь тенденциями развития новых информационных технологий. Книга очень полезна студентам факультетов вычислительной техники, прикладной математики, вычислительной математики и кибернетики, т. е. факультетов, которые готовят потенциальных системотехников, программистов. Руководителям программных проектов будет полезно ознакомиться с предложенными автором подходами к распределению ресурсов при групповой разработке программ; системотехникам — с подходами к описанию систем; программистам-практикам — с подходами к эффективному использованию объектных и объектно-ориентированных языков программирования. Студенты получат конкретные инструкции для новичков, ознакомятся со спецификой объектного мышления; получат начальную информацию о проектировании сложных систем, языках программирования и сложностях, связанных с применением объектно-ориентированного подхода к проектированию; получат хороший урок, как надо представлять и оформлять информацию для читателя, как работать с литературой; ознакомятся с именами всех наиболее популярных разработчиков объектно-ориентированных программных продуктов и многое другое.

Для всех категорий читателей будет полезно ознакомиться с видами проявления сложности, рассмотреть объектно-ориентированный подход как средство решения проблемы сложности, ознакомиться с основными элементами подхода и основными понятиями. Очень важно описание самой методологии разработки сложных систем с применением объектно-ориентированного подхода, системы графического отображения для документирования, управления процессом проектирования. В книге метод объектно-ориентированного проектирования рассмотрен в непосредственной связи с традиционными методами, объективно представлены его достоинства и недостатки. Ценность раздела Применение — демонстрация на примерах, как объектно-ориентированный подход помогает решать проблему сложности в больших системах. Такие фирмы, как AT @ T, Boeing, General Electric, IBM, Philips, Rockwell International, Shell Oil и другие, менее известные у нас, предоставили автору тексты программ из программных систем, которые использованы им в книге в качестве примеров. Кроме небольших примеров, поясняющих отдельные элементы

объектно-ориентированной технологии, целый раздел книги посвящен примерам промышленного применения объектно-ориентированных языков программирования Smalltalk, Object Pascal, C++, CLOS и Ada. В начале каждого примера этого раздела автором поясняется специфика предметной области, для которого создавалась программная система, описанная в примере. В приложении приведены сравнительные характеристики различных объектных и объектно-ориентированных языков программирования.

Организация книги продумана автором таким образом, чтобы читателю было удобно ориентироваться в интересующих его вопросах. Для удобства в конце каждой главы введено заключение, есть ссылки на дополнительную литературу. Книга написана живым языком. Поражает, с какой скрупулезностью автор рассматривает каждый из этапов объектно-ориентированной технологии и с какой любовью и уважением он относится к своему читателю, предоставляя всю информацию по предметной области, которую можно было разместить в книге объемом 580 страниц. Книга признана в США одной из наиболее покупаемых по объектно-ориентированному подходу в 1991 году. С выходом перевода книги появляется редкая, если не уникальная, возможность изучать объектно-ориентированное проектирование по тому же источнику, каким пользуются сегодня в Европе и США. Я надеюсь, что у книги будет долгая жизнь.

Об Авторах перевода и подготовки книги к изданию. Перевод книги осуществлен Александром Ивановым, Александром Карнауковым, Михаилом Шелкиным. Все переводчики имеют опыт использования объектно-ориентированного подхода в разработке программных систем. В переводе Глав 4, 5 и 12 участвовали специалисты ОНИЦ ПЭВМ «Техно» М. В. Кузнецов и И. В. Меренкова. Основная работа по подготовке оригинал-макета книги проведена Сергеем Деревянко. В подготовке и редактировании перевода отдельных глав участвовала Наталия Ближнюк.

Я уверен, что практический интерес к объектно-ориентированной технологии проектирования будет нарастать по мере появления отечественных наукоемких программных продуктов и систем, которые будут конкурировать с западными. Особое значение будет иметь использование объектно-ориентированной технологии при совместной разработке сложных проектов интернациональными командами разработчиков. Появление книги сегодня может сыграть важную роль в развитии объектно-ориентированной технологии в нашей стране.

Желаю Вам удачи
Андрей Артамошкин
18 апреля 1992

Введение

Бог даровал человеку жажду духовного мира, эстетических ощущений, безопасности, справедливости и свободы, которые нельзя обеспечить простым промышленным производством. Но производство позволяет уйти от постоянной борьбы с нищетой и приблизиться к всеобщему благосостоянию, высвободить дополнительное время для духовного и эстетического развития, а также переложить часть проблем на специально создаваемые для этой цели институты религии, закона и охраны свободы.

Харлан Миллс
(DPMA and Human Productivity)

Как профессионалы в вычислительной технике, мы стремимся создавать полезные и работающие системы; а как программисты, мы вынуждены создавать сложные системы в условиях сильно ограниченных ресурсов. В последние несколько лет объектно-ориентированная технология развивалась в различных областях вычислительной техники как средство решения проблем, связанных со сложностью создаваемых систем. Объектный подход доказал свою эффективность и универсальность.

Цели

Так как опыт применения объектно-ориентированного проектирования относительно невелик, об эффективном использовании элементов объектного подхода, как дисциплины, говорить еще рано. Данная книга — первое практическое руководство по созданию объектно-ориентированных систем. Автор ставил перед собой следующие цели:

- * Разъяснить фундаментальные концепции объектного подхода.
- * Помочь овладеть терминологией и методами объектно-ориентированного проектирования.
- * Научить применять на практике объектно-ориентированное проектирование, используя языки программирования Smalltalk, Object Pascal, C++, CLOS и Ada.

Все концепции, рассмотренные в данной книге, имеют глубокую теоретическую основу; тем не менее это — книга для практического использования, в которой обсуждаются проблемы создания больших промышленных программных систем.

К читателю

Книга рассчитана на профессионалов и студентов, начинающих изучать данные вопросы.

- * Программисты-практики найдут подходы к эффективному использованию объектных и объектно-ориентированных языков программирования при решении практических задач.
- * Аналитики и системные проектировщики найдут рекомендации по переходу от исходных требований к их реализации с помощью методологии объектно-ориентированного проектирования, научатся отличать «хорошие» проекты от «плохих» и производить изменения, вызванные внешними причинами, а также, что наиболее существенно, найдут новые подходы к реализации сложных систем.

- * Руководители программных проектов могут ознакомиться с приемами распределения ресурсов при групповой разработке программ и способами преодоления трудностей, связанных с созданием сложных программных систем.
- * Разработчики и пользователи инструментальных средств найдут подробную информацию по системе обозначений, применяемой при документировании процесса объектно-ориентированного проектирования.
- * Студенты получают инструкции, позволяющие сделать первые шаги в теории и практике разработки сложных программных систем.

Книгу можно использовать как учебное пособие в вузах и для самостоятельного изучения предмета. Книга состоит из трех основных частей: концепции, методология и применения.

Концепции

В части I рассмотрены вопросы, связанные со сложностью программных систем и видами проявления этой сложности. Объектный подход предлагается в качестве средства решения проблемы сложности. Подробно рассматриваются основные элементы объектного подхода: абстракции, ограничение доступа, модульность, иерархия, типирование, параллелизм и устойчивость. Определяются такие понятия, как класс и объект. Поскольку идентификация классов и объектов является ключевым моментом в объектно-ориентированном проектировании, в книге уделено достаточное внимание изучению природы классификации. Анализируются, в частности, подходы к классификации в таких областях знаний, как биология, лингвистика и психология; затем эти подходы применяются к программным системам с использованием методов анализа предметной области и концептуального моделирования.

Методология

В части II описана методология разработки сложных систем на основе объектного подхода, названного объектно-ориентированным проектированием OOD (object-oriented design). Представлена система графических обозначений для документирования процесса OOD, в частности особенности отдельных этапов жизненного цикла программных систем и управление процессом проектирования.

Применения

В части III рассмотрены пять оригинальных примеров использования методологии OOD в различных предметных областях. Эти примеры представляют основные разновидности сложных проблем, с которыми сталкиваются разработчики программных средств на практике. Показать то, как отдельные принципы реализуются в практических задачах, не сложно, но гораздо важнее показать, как объектный подход помогает решать проблемы сложности в больших системах. Это позволяет сфокусировать внимание на способах построения действительно полезных систем, решающих актуальные задачи. Поскольку выбранные предметные области знакомы не для всех читателей, рассмотрение каждого примера начинается с краткого обзора его особенностей (например, структуры базы данных или архитектуры типа «классной доски»). Автор показывает с помощью примеров, как при объектно-ориентированном проектировании могут быть использованы различные объектные и объектно-ориентированные языки программирования: Smalltalk, Object Pascal, C++, CLOS и Ada. Разработка программных систем редко

выполняется по методу поваренной книги, поэтому особо выделены последовательный интерактивный процесс создания системы с множеством четко обозначенных критериев и глубоко проработанных подходов.

Дополнительная информация

Все разделы книги содержат необходимый дополнительный материал. В большинстве глав специально выделена справочная информация по наиболее важным вопросам, например механизм реализации методов в различных объектно-ориентированных языках программирования. В книгу включено приложение, в котором приведены сравнительные данные по используемым объектным и объектно-ориентированным языкам и основные характеристики языков программирования.

Для читателей, не владеющих используемыми в практических примерах языками, приведены основные конструкции этих языков и соответствующие примеры программ. Книга содержит толковый словарь по наиболее употребительным терминам и понятиям, а также обширную библиографию. Примеры программ для гл. 8 — 12 предоставлены издателем.

Работа с книгой

Книга предназначена для чтения «от корки до корки», а не по частям. Если вы хотите глубоко понять принципы объектного подхода или четко усвоить процедуры OOD, следует начать чтение гл. 1 и продолжать далее по порядку. Если вас в большей степени интересуют особенности процесса OOD и его документирование, можно начать с гл. 5, 6. Гл. 7 особенно полезна для руководителей программных проектов, использующих объектную методологию. Для тех, кто интересуется примерами практического применения OOD в конкретной предметной области, предназначены гл. 8 — 12.

Благодарности

Книгу посвящаю моей жене Jan за ее любовь и поддержку.

Мои представления об объектно-ориентированном проектировании были сформированы в результате множества дискуссий и обмена по обычной и электронной почте с целым рядом специалистов. За вклад в создание книги я приношу благодарность Sid Bailin, Daniel Bobrow, Dick Bolz, Dave Bernstein, Brad Cox, Tom DeMarco, Mike Devlin, Adele Goldberg, Tony Hoare, Michael Jackson, Ralph Johnson, James Kemph, Phil Levy, Barbara Liskov, James MarcFarlane, Masoud Milani, Harlan Mills, Steve Neis, Dave Parnas, Bill Riddel, Kurt Schmucker, Ed Seidewitz, Dan Shiffman, Bjarne Stroustrup, Dave Thomas, Mike Vilot, Tony Wasserman, Peter Wegner, Lloyd Williams, Niklaus Wirth, Ed Yourdon. Большая часть практических приемов получена автором из множества больших программных систем, созданных такими известными компаниями, как AT&T, Autotrol, Boeing, Computer Sciences Corporation, Contel, Ericsson, Ferranti, General Electric, GTE, Holland Signaal, Hughes Aircraft Company, IBM, Lockheed, Martin Marietta, NTT, Philips, Rockwell International, Shell Oil, TRW. В работе над книгой согласились помочь сотни профессиональных программистов и руководителей проектов, и я благодарю их за то, что они помогли приблизить содержание книги к современным реальным проблемам.

Особую благодарность я выражаю редактору Alan Apt за постоянную поддержку в работе над книгой. Благодаря Tony Hall и его карикатурам книга отличается от традиционно скучных технических книг. Автор благодарен также трем своим котам (Samy, Annie, Shadow), составлявшим ему компанию поздними вечерами при написании книги.

Часть I

КОНЦЕПЦИИ

Сэр Исаак Ньютон однажды признался своим друзьям, что хотя он и открыл закон гравитации, но не знает механизма его работы.

Лили Томлин (Lily Tomlin)

The Search for Signs of Intelligent Life in the Universe

Глава 1

Сложность

Врач, инженер-строитель и программист спорили о том, чья профессия древнее. Врач заметил: «В Библии сказано, что Бог сотворил Еву из ребра Адама. Такая операция может быть проведена только хирургом, поэтому я по праву могу утверждать, что моя профессия самая древняя в мире». Тут вмешался строитель и сказал: «Но еще раньше Бог сотворил небо и землю из хаоса. Это первое и, несомненно, наиболее выдающееся применение строительной инженерии. Поэтому, дорогой доктор, Вы неправы. Моя профессия самая древняя в мире». Программист при этих словах откинулся в кресле, загадочно улыбнулся и произнес: «Да, но кто, как вы думаете, сотворил хаос?»

1.1. СЛОЖНОСТЬ, ПРИСУЩАЯ ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

Простые и сложные программные системы

«Умирающая» звезда в предверии коллапса, ребенок, который учится читать, клетки крови, атакующие вирус, — все это объекты физического мира, обладающие очень большой степенью сложности. Программы для ЭВМ могут также содержать элементы, однако их сложность совершенно другая. Брукс [1] пишет: «Эйнштейн утверждал, что должно существовать простое объяснение всех природных процессов, так как Бог не капризен и при нем невозможен произвол. Вера в это не может утешить программиста: сложность создаваемых им программ целиком зависит от него самого».

Очевидно, не все программные системы являются сложными. Существует много программ, задуманных, разработанных, сопровождаемых и используемых одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий в отрыве от коллег; но о таких программах все быстро забывают и не о них идет сейчас речь. Мы не хотим сказать, что все такие системы плохо сделаны, или тем более усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их предпочитают заменить новым программным обеспечением, чем

постоянно пытаться переделывать или расширять их функциональные возможности. Разрабатывать подобные прикладные пакеты скорее утомительно, чем сложно, и мы мало заинтересованы в изучении того, как это делать.

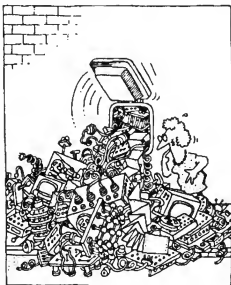
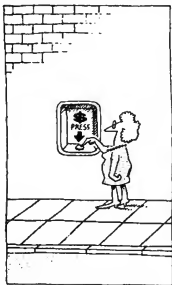
Зато нас больше интересует проблема разработки того, что я буду называть индустриально организованными программными продуктами. Они применяются для решения самых разных задач, таких, например, как системы с обратной связью, которые контролируют ход определенного физического процесса и для которых критическими параметрами являются время обработки и свободное пространство памяти; задачи поддержания целостности больших объемов информации при обеспечении к ним параллельного доступа различных пользователей; системы управления и контроля за реальными процессами (линии воздушного или железнодорожного сообщения). Системы такого типа обычно имеют большое время жизни и обслуживают большое количество пользователей, результат работы которых во многом зависит от нормального функционирования предложенных программных средств. Индустриально организованное программное обеспечение помогает решать задачи моделирования таких сложных процессов, как планирование оптических экспериментов и имитация определенных сторон человеческого мышления. Существенной чертой индустриально организованных программных средств является их большая сложность: практически невозможно охватить все тонкости системы одним разработчиком. Грубо говоря, сложность таких систем превышает возможности человеческого интеллекта. Увы, но сложность, о которой мы говорим, является, по-видимому, необходимым свойством всех больших программных систем. Под необходимостью мы имеем в виду следующее: можно создать эту сложность, но нельзя придумать так, чтобы обойтись без нее.

Конечно, среди нас всегда есть гении, люди экстраординарных возможностей, которые в одиночку смогут сделать работу за группу обычных людей-разработчиков и добиться в своей области успеха, сравнимого с достижениями Леонардо да Винчи. Такие люди нам нужны как архитекторы систем, т.е. те, кто изобретает новые механизмы, которыми можно пользоваться как базовыми при разработке других систем и решении других задач. Однако, как замечает Питерс: «В мире очень мало гениев. И не надо думать, будто в среде инженеров-программистов их пропорция слишком велика» [2]. Несмотря на то что все мы чуточку гениальны, в области индустриально организованного программирования нельзя постоянно полагаться на божественное вдохновение, которое обязательно поможет нам. Поэтому мы должны прежде всего рассмотреть уже известные способы конструирования сложных систем. Для лучшего понимания того, с чем мы собираемся иметь дело, сначала ответим на вопрос: почему сложность присуща всем большим программным системам?

Почему программному обеспечению присуща сложность?

Как говорит Брукс: «Сложность программного обеспечения — отнюдь не случайное его свойство, скорее необходимое» [3]. Его сложность определяется четырьмя основными причинами: сложностью проблемы, сложностью управления процессом разработки, сложностью обеспечения гибкости конечного программного продукта и сложностью описания поведения отдельных подсистем.

Сложность проблемы. Проблемы, которые мы пытаемся решить с помощью разрабатываемого программного обеспечения, часто неизбежно содержат сложные элементы, к которым предъявляется множество различных, нередко противоположных требований. Рассмотрим требования к электронной системе самолета с несколькими двигателями, к телефонной коммутаторной системе или к автономному роботу. Довольно трудно даже в общих чертах понять, как работает данная система, но прибавьте к этому все (часто неявные) дополнительные требования, такие, как удобство, производительность, цена, выживаемость и надежность! Эта высокая степень сложности задачи и порождает ту сложность программного продукта, о которой пишет Брукс. Внешняя сложность обычно возникает из-за «несоответствия импедансов», которое существует между пользователями системы и ее разработчиками: пользователи обычно с трудом могут внятно объяснить разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны, просто каждая из групп является экспертом лишь в своей области и ей недостает знаний в области партнера. У пользователей и разработчиков разные взгляды на сущность проблемы и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены в больших по объему текстах, слегка «разбавленных» немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.



Задачей разработчиков программного обеспечения является создание иллюзии простоты.

Дополнительная сложность обуславливается изменением требований к программной системе уже в процессе разработки в основном из-за того, что само существование проекта программной системы часто изменяет проблему. Рассмотрение первых результатов — схем, прототипов — и затем использование системы уже после того, как она разработана и установлена, заставляют пользователей самим лучше понять и отчетливее сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в области, связанной с решением самой проблемы, и позволяет им задавать более осмысленные вопросы, касающиеся поведения системы.

Большая программная система — объект капиталовложений, и мы не можем себе позволить создавать ее заново каждый раз при изменении требований к ней. Поэтому, хотим мы этого или нет, большие системы имеют тенденцию к эволюции в процессе их использования — к тому, что часто неправильно называют программным сопровождением. Если быть более точным, то сопровождение имеет в виду устранение ошибок; эволюция подразумевает внесение изменений в систему в ответ на изменившиеся требования к ней; сохранение — использование всех возможных и невозможных способов для обеспечения функционирования старой и давно утратившей пригодность системы. К сожалению, опыт показывает, что существенный процент затрат на разработку программных систем тратится именно на сохранение.

Сложность управления процессом разработки. Основная задача разработчиков состоит в создании иллюзии простоты, защищающей пользователей от сложности описываемого предмета или процесса. Размер исходных текстов программной системы отнюдь не входит в число ее главных достоинств, поэтому исходные тексты стараются сделать более компактными, используя при этом уже существующие методы и изобретая новые, более мощные. Однако большое число требований к системе иногда неизбежно приводит либо к необходимости создания нового программного продукта значительных размеров, либо к модификации существующего, что также не делает его проще. Всего 20 лет назад программы объемом в несколько тысяч строк, написанные на ассемблере, выходили за пределы инженерных возможностей. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионов строк (и к тому же на языках высокого уровня). Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложили ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, хотя в идеале численность этой команды должна быть как можно меньшей. Но какой бы она ни была, всегда возникнут значительные трудности, связанные с координацией хода работ над таким коллективным продуктом. Большее число разработчиков подразумевает более сложные связи между ними, особенно если участники работ географически удалены друг от друга — ситуация, типичная при реализации больших проектов. Таким образом, в случае коллективного проекта главной проблемой руководства является поддержание целостности основной идеи в ходе работ.

Гибкость программного обеспечения. Строительные компании обычно не имеют деревообрабатывающих заводов для производства строительной древесины; было бы также странно, если бы на месте строительства

сооружались прокатные станы для ковки стальных балок под будущее здание. Однако в программной индустрии такая практика — дело обычное. Программирование обладает максимальной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость — чрезвычайно соблазнительное качество. Оно, однако, заставляет разработчика создавать самому все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких абстрактных уровней. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и на качества материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень кропотливым делом.

Сложность описания поведения отдельных подсистем. Когда мы кидаем вверх мяч, мы можем довольно надежно предсказать его траекторию, потому что знаем все силы, действующие на него в нормальных условиях. Мы бы очень удивились, если бы, кинув мяч с чуть большей скоростью, увидели, что он на середине пути неожиданно остановился и стремительно рванулся вертикально вверх¹⁾. В недостаточно отлаженной программе моделирования полета мяча такая ситуация легко может возникнуть.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько способов контроля за ними. Полный список этих переменных, их текущих значений, текущих адресов и стеков описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как системы моделирования движения мяча, напротив, являются непрерывными. Парнас пишет, что «когда мы говорим, что система описывается непрерывной функцией, это означает, что она не содержит никаких сюрпризов. Небольшие изменения входных параметров всегда вызовут соответственно небольшие изменения выходных» [4]. С другой стороны, дискретные системы по определению имеют конечное число возможных состояний; в больших системах это число в соответствии с правилами комбинаторики достигает громадных значений. Мы стараемся спроектировать системы так, чтобы поведение одной части системы оказывало минимальное воздействие на поведение другой. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты взаимодействий между событиями. Представим себе пассажирский самолет, в

1) Наличие хаоса может усложнить поведение даже простых систем. Невозможно точно предугадать состояние системы, если оно зависит от случайных факторов. Зная, например, начальное положение двух капель воды в потоке, мы не можем точно указать, на каком расстоянии друг от друга окажутся эти капли через некоторое время. От случайных воздействий зависит поведение практически всех природных систем: атмосферы, химических молекул, биологических систем, и даже компьютерных сетей. Хотя, по-видимому, даже в хаотичных системах существуют некие скрытые законы, подтверждение тому — тенденция к возникновению аттракторов.

котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно стать свидетелем ситуации, когда результатом нажатия кнопки включения головного света одним из пассажиров стал бы немедленный ввод самолета в глубокое пикирование. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что, исключая, возможно, наиболее тривиальные случаи, всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, при определении ее точности мы должны удовлетвориться существующими уровнями доверия к системе.

Трудности на пути создания сложных систем

«Чем сложнее система, тем ближе она к полному развалу» [5]. Строителям редко приходят мысли об установке дополнительного фундамента к уже построенному 100-этажному небоскребу; сделать это будет совсем недешево, да и, наверняка, ничего не выйдет. Но что удивительно, пользователи программных систем совсем нередко ставят подобные задачи перед разработчиками. Это, утверждают они, всего лишь технический вопрос для программистов.

Наши неудачные попытки создать сложную программную систему реализуются в проектах, которые выходят за пределы установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям. Мы часто называем это кризисом программного обеспечения, но, честно говоря, болезнь, которая продолжается слишком долго, уже не болезнь, она становится частью нормального процесса. К сожалению, этот кризис приводит к разбазариванию человеческих ресурсов — самого драгоценного товара — и к существенному ограничению возможностей создания нового продукта. Сейчас просто не хватает достаточного количества хороших программистов, чтобы обеспечить всех пользователей нужными им программами. Более того, существенный процент персонала, занятого разработками в любой организации, часто должен заниматься в основном сопровождением и сохранением устаревших программ. Принимая во внимание явный и неявный вклад, вносимый индустрией программного обеспечения в развитие экономической базы большинства ведущих стран, и анализируя пути усиления с помощью программных средств возможностей индивидуума, становится ясной невозможность сохранения существующей схемы создания программных систем.

Как мы можем изменить эту ситуацию? Так как проблема возникает в целом в результате сложности структуры программных продуктов, мы предлагаем сначала изучить способы организации сложных структур в других научных дисциплинах. В самом деле, можно привести множество примеров успешно функционирующих сложных систем. Некоторые из этих систем созданы человеком, такие, как, например, Спейс Шаттл, англо-французский туннель, или большие фирмы типа IBM. В природе существуют еще более сложные системы, например система обмена веществ у человека или структура растения.

1.2. СТРУКТУРА СЛОЖНЫХ СИСТЕМ

Примеры сложных систем

Персональный компьютер (ПК). Персональный компьютер — прибор средней степени сложности. Большинство ПК состоит из одних и тех же основных элементов: центрального процессора (ЦП), монитора, клавиатуры и запоминающего устройства какого-либо типа (либо гибкого диска, либо встроенного жесткого диска). Мы можем взять любую из этих частей и разложить ее в свою очередь на составляющие. ЦП, например, обычно содержит первичную память, арифметико-логическое устройство (АЛУ) и шину, к которой подключены периферийные устройства. Каждую из этих частей можно также разложить на составляющие: АЛУ состоит из регистров и из логики произвольного управления, которые сами состоят из еще более простых элементов типа логическое «И», инверторы и т.д. Это пример иерархической структуры сложной системы. Персональный компьютер своей нормальной работой обязан четкому совместному функционированию каждой из его составных частей. В самом деле, мы можем понять, как работает компьютер, только когда отдельно рассмотрим каждую его составляющую. Таким образом, мы можем изучать устройства монитора и жесткого диска независимо друг от друга. Аналогично мы можем изучать АЛУ, не рассматривая при этом подсистему первичной памяти.

Сложные системы не просто иерархичны: уровни их иерархии отражают различные уровни абстракции, вытекающие друг из друга, но обладающие при этом определенной степенью автономности. И для каждой конкретной задачи мы рассматриваем соответствующий уровень. Например, если бы мы пытались решить проблему синхронизации с помощью первичной памяти, нам предстояло бы аккуратно рассмотреть архитектуру соответствующего уровня, но этот уровень абстракции был бы неприемлем, если мы старались найти решение в самой прикладной программе.

Животные и растения. Ботаник старается найти похожие и отличительные черты у разных растений, изучая их морфологию, т.е. их форму и структуру. Растения — это сложные многоклеточные организмы. В результате совместной деятельности различных систем и органов растения возникают такие сложные процессы, как фотосинтез и обмен веществ с окружающей средой. Растения состоят из трех основных структур (корни, ствол, стебли и листья), и каждая из них имеет свое устройство. Корни, например, состоят из ветвей, волос, вершин и шапки. Если же сделать разрез листа, то мы увидим, что он состоит из эпидермиса, мезофилла и сосудистой ткани. Каждая из этих структур в свою очередь представляет собой набор клеток. Внутри каждой клетки существует следующий уровень сложности, который включает хлоропласты, митохондрию, ядро и т.д. По аналогии со структурой компьютера части растения формируют иерархию, и каждый уровень этой иерархии отражает свой уровень сложности.

Все составляющие одного уровня абстракции взаимодействуют друг с другом вполне определенным способом. Если, например, рассмотреть высший уровень абстракции, то корни отвечают за доставку воды и минералов из почвы. Корни взаимодействуют со стволами и стеблями, по которым эти материалы достигают листьев. Листья в свою очередь используют воду и минералы, полученные через стебли, для производства пищи с помощью фотосинтеза.

На каждом уровне существуют четкие границы между внешней и внутренней средой. Мы можем сказать, что части листа работают вместе для обеспечения функционирования листа как целого, и почти не осуществляют взаимодействия с элементами корней. Проще говоря, существует четкое разделение функций между элементами на различных уровнях абстракции.

В компьютере логические схемы «И» используются и в конструкции ЦП, и в конструкции жесткого диска. Аналогично большое число «унифицированных элементов» имеется во всех частях структурной иерархии растения. Это естественный путь для достижения экономии средств выражения. Например, клетки служат основными строительными блоками всех структур растения; в конечном счете корни, стебли и листья растения состоят из клеток. Но при этом существует много различных типов клеток: например, клетки, содержащие и не содержащие хлоропласты, клетки со стенками, которые пропускают или, наоборот, не пропускают воду, и даже живые и неживые клетки.

При изучении морфологии растения невозможно выделить в нем отдельные части, каждая из которых выполняет лишь одну, вполне определенную функцию сложного процесса, например фотосинтеза. Также фактически не существует отдельных элементов, напрямую координирующих деятельность подсистем растения. Отдельные его части действуют достаточно сложным образом в качестве независимых агентов, совместное функционирование которых полностью определяет поведение системы. Таким образом, поведение растения обеспечивается суммой бессмысленных на первый взгляд действий этих агентов.

Структура строения многоклеточных животных также является иерархической: клетки формируют ткани, ткани работают вместе как органы, группы органов определяют системы (пищеварительную, например) и так далее. Природа умеет создавать системы, отличающиеся высокой степенью универсальности базовых элементов: основной строительный блок всех растений и животных — клетка. Естественно, между этими двумя типами клеток существуют различия. Клетки растения, например, заключены в твердую целлюлозную оболочку в отличие от клеток животных. Но, несмотря на данные различия, обе указанные структуры, несомненно, являются клетками. Это удивительный пример схожести строения систем, относящихся к разным сферам биологической деятельности.

То же самое можно сказать и о некоторых механизмах надклеточного уровня. И растения, и животные используют сосудистую систему для транспортировки питательных веществ. И у тех и у других может существовать различие полов внутри одного вида.

Материя. Исследования в таких разных областях, как астрономия и ядерная физика, дают нам много примеров очень сложных систем. Рассмотрев эти две дисциплины, мы найдем дополнительные примеры структурной иерархии. Астрономы занимаются изучением галактик и их составляющих: звезд, планет и других небесных тел. Ядерные физики имеют дело со структурной иерархией физических тел совсем другого масштаба. Атомы состоят из электронов, протонов и нейтронов; электроны, по-видимому, являются элементарными частицами, но протоны, нейтроны и другие частицы формируются из еще более мелких компонент, называемых кварками.

Однако эти две совершенно различные по масштабу и структуре физические системы подчиняются одним и тем же законам взаимодействия. На самом деле оказывается, что во Вселенной существуют всего четыре типа сил: гравитационная, электромагнитная, сильное взаимодействие и слабое взаимодействие. И многие законы физики, подразумевающие наличие этих элементарных сил, такие, как закон сохранения энергии и момента, можно применить и к галактикам, и к кваркам.

Общественные институты. Как последний пример сложных систем, давайте рассмотрим структуру общественных институтов. Группы людей собираются вместе для решения задач, которые не могут быть решены отдельными личностями. Некоторые организации быстро распадаются, а некоторым удается продержаться на протяжении нескольких человеческих жизней. Чем больше организация, тем отчетливее видна в ней иерархическая структура. Мультинациональные корпорации состоят из компаний, которые в свою очередь состоят из подразделений, содержащих различные филиалы. Последним принадлежат уже отдельные офисы и т.д. На протяжении существования организации границы между этими частями могут изменяться, так что с течением времени старая иерархия путем эволюции трансформируется в новую, более стабильную.

Отношения между разными частями большой организации такие же, как и между компонентами компьютера, растения, галактики. Таким образом, степень взаимодействия между сотрудниками одного учреждения, несомненно, выше, чем между сотрудниками двух разных учреждений. На первый взгляд это не совсем так: почтовый клерк, например, обычно не взаимодействует с исполнительным директором компании, в основном обслуживая посетителей. Однако клерка и директора, принадлежащих разным уровням иерархии, объединяет общий механизм функционирования компании. Работа и клерка, и директора оплачивается одной финансовой организацией, и оба они пользуются общей аппаратурой, в частности внутренней телефонной системой компании для решения своих задач.

Пять признаков сложной системы

Основываясь на работе Саймона и Эндо, Куртуа предлагает следующие пять признаков сложной системы.

1. «Сложность часто представляется в виде иерархии. Сложная система обычно состоит из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самых низших уровней абстракции» [6].

Тот факт, что многие сложные системы имеют разложимую на составляющие иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже «увидеть» такие системы и их подсистемы [7]. В самом деле, скорее всего мы сможем понять лишь те системы, которые имеют иерархическую структуру.

2. Выбор низшего уровня абстракции достаточно произволен и в большой степени определяется наблюдателем.

Низший уровень для одного наблюдателя может оказаться уровнем достаточно высокой абстракции для другого.

Саймон называет иерархические системы «разложимыми», если они могут быть разделены на четко узнываемые части, и «почти разложимыми», если их составляющие не являются абсолютно независимыми.

3. «Внутриэлементные связи обычно сильнее межэлементных связей. Поэтому высокочастотные взаимодействия внутри структуры оказываются естественным образом отделены от низкочастотных взаимодействий между структурами» [8].

Различие между внутри- и межэлементными взаимодействиями обуславливает разделение системы на абстрактные автономные части, которые можно изучать по отдельности.

Как мы уже говорили, многие сложные системы организованы достаточно экономно в смысле способов выражения. Отсюда — следующий признак сложных систем.

4. «Иерархические системы обычно состоят из нескольких подсистем разного типа, реализованных в разном порядке и в разнообразных комбинациях» [9].

Иногда (например, клетки растений и животных) можно обнаружить подсистемы, общие для различных сфер функционирования всей системы.

Выше мы говорили, что сложные системы имеют тенденцию к развитию во времени. Саймон считает, что сложные системы будут развиваться из простых гораздо быстрее, если для них существуют устойчивые промежуточные формы [10].

5. «Работающая сложная система неизбежно оказывается результатом развития работающей простой системы... Сложная система, разработанная от начала до конца на бумаге, никогда не работает и нельзя заставить ее заработать. Вы должны начать с работающей простой системы» [11].

В процессе развития системы объекты, которые сначала считаются сложными, начинают рассматриваться как элементы низших уровней абстракции, из которых затем строятся более сложные системы.

Организованная и неорганизованная сложность

Каноническая форма сложной системы. Обнаружение общих абстракций и их механизмов значительно облегчает понимание сложных систем. Опытный пилот, например, сориентировавшись всего за несколько минут, может взять на себя управление тяжелым реактивным самолетом, на котором он раньше никогда не летал, и спокойно его вести. Определив элементы, общие для всех подобных самолетов (такие, как руль направления, элероны и механизм управления тягой), пилот прежде всего должен найти отличия этого самолета от других. Если пилот уже знает, как управлять данным самолетом, ему гораздо легче научиться управлять самолетом, похожим на него.

В этом примере мы использовали термин «иерархия» в довольно широком смысле. Наиболее интересные системы содержат много разных иерархий. В самолете, например, можно выделить системы силовой установки, управления полетом и т.д. Такое разбиение дает структурную иерархию типа «это — часть того». Но одновременно эту же систему можно рассмотреть по-другому. Например, турбореактивный двигатель — особый тип реактивного двигателя, а «Pratt and Whitney TF30» — особый тип турбореактивного двигателя. Определенный другим путем «реактивный двигатель» представляет обобщение свойств, присущих любому типу реактивного двигателя; турбореактивный двигатель — это просто особый тип реактивного двигателя со свойствами, которые отличают его, например, от прямоточного двигателя. Эта вторая иерархия представляет собой «типовую» иерархию. Исходя из нашего опыта, мы сочли необходимым рассмотреть

систему с двух точек зрения, выделив типовую и структурную иерархии. В гл. 2 мы назовем эти иерархии соответственно структурой классов и структурой объектов.

Если мы объединим понятия структуры классов и структуры объектов с пятью признаками сложных систем, мы найдем, что фактически все сложные системы можно представить одной и той же (канонической) формой, представленной на рис. 1-1. Здесь мы видим две разные иерархии, принадлежащие одной системе: структуру классов и структуру объектов. Каждая иерархия является многоуровневой, в которой более абстрактные классы и объекты построены на базе более примитивных. Выбор класса или объекта, соответствующего низшему уровню абстракции, зависит от конкретной задачи. Среди объектов одного уровня существуют четко выраженные связи, особенно это касается компонентов структуры объектов. Любому рассматриваемому уровню абстракции соответствует свой уровень сложности. Заметьте также, что структуры классов и объектов не являются независимыми: каждый объект в структуре объектов представляет определенный класс. Как видно из рис. 1-1, объектов в сложной системе обычно гораздо больше, чем классов. Если бы мы не показали «классовую структуру» нашей системы, нам пришлось бы дублировать знания о свойствах каждой ее отдельной части. С введением данной структуры мы размещаем эти основные свойства в одном месте.

Обычно наиболее успешными программными системами являются те, в которых заложены хорошо продуманные структуры классов и объектов и которые обладают пятью признаками сложных систем, описанными выше. Оценим важность этого наблюдения и выразимся более категорично: очень редко можно встретить программную систему, разработанную точно по графику, уложившуюся в бюджет и удовлетворяющую требованиям заказчика, в которой бы не были учтены эти особенности.

Человеческие возможности и сложные системы. Если мы знаем, какой должна быть конструкция сложных программных систем, то почему при разработке таких систем мы сталкиваемся с серьезными проблемами? Как показано в гл. 2, понятие организованной сложности программ (основные принципы создания которой мы будем обозначать понятием «объективный подход») относительно ново. Существует, однако, еще одна, по-видимому, главная причина: ограниченность биологических возможностей человека.

Когда мы начинаем анализировать сложную систему, в ней обнаруживается много составных частей, которые взаимодействуют друг с другом различными сложными способами, причем и сами части системы, и пути их взаимодействия не обнаруживают никакого сходства. Это пример неорганизованной сложности. Когда мы начинаем в процессе проектирования вносить в систему элементы организованности, мы должны думать сразу о многих вещах. Например, в системе управления воздушным движением необходимо одновременно контролировать состояние многих летательных аппаратов, в том числе такие их параметры, как положение, скорость и направление полета. При анализе дискретных систем необходимо рассматривать большие, сложные и не всегда детерминированные пространства состояний. К сожалению, один человек не может отслеживать все это одновременно. Психологи (например, Миллер) считают, что максимальное количество единиц информации, которое человеческий мозг может одновременно обработать, не превышает 7 [12]. Этот объем связан,

по-видимому, с объемом краткосрочной памяти у человека. Саймон отмечает также, что дополнительным ограничивающим фактором является скорость обработки мозгом поступающей информации: ему требуется примерно 5 с на каждое новое событие [13]. Таким образом, мы столкнулись с серьезным препятствием: требуемая сложность программных систем возрастает, а способности нашего мозга контролировать эту сложность остаются на прежнем уровне. Как нам выйти из данного затруднительного положения?

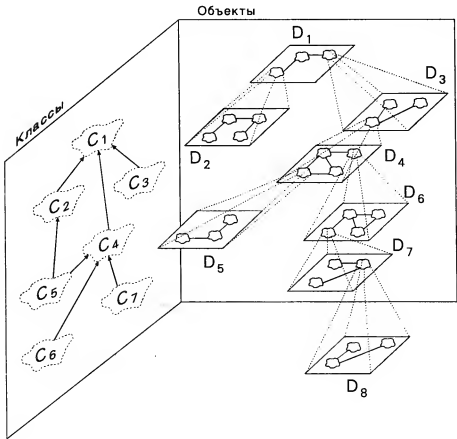


Рис. 1-1. Каноническая форма сложной декомпозиции.

1.3. ВНЕСЕНИЕ ПОРЯДКА В ХАОС

Декомпозиция

«Способ управления сложными системами был известен еще в древности: *divide et impera* (разделяй и властвуй)» [14]. При проектировании сложной программной системы необходимо составлять ее из небольших подсистем, каждую из которых можно отладить независимо от других. В этом случае мы не выходим за пределы возможностей человека, отпущенных ему природой: при разработке любого уровня системы нам необходимо будет одновременно держать в уме информацию лишь о немногих ее частях (отнюдь не о всех). В самом деле, правильная декомпозиция непосредственно определяет сложность, присущую программной системе, обеспечивая разделение пространства состояний системы [15].

Алгоритмическая декомпозиция. У большинства из нас понятие «проектирование» ассоциируется со структурным проектированием по методу «сверху вниз», и мы воспринимали декомпозицию как обычное разделение алгоритмов, где каждый модуль системы выполняет один из важных этапов общего процесса. На рис. 1-2 приведен пример результата структурного проектирования: структурная схема, которая иллюстрирует связи между различными функциональными элементами системы.

Данная структурная схема отражает конструкцию программной подсистемы, осуществляющей изменение содержания управляющего файла. Она была автоматически получена путем анализа диаграммы потоков данных экспертной системой, которой известны правила структурного проектирования [16].

Объектно-ориентированная декомпозиция. Мы полагаем, что существует альтернативный способ декомпозиции этой подсистемы. На рис. 1-3 мы разделили подсистему, выбрав в качестве критерия декомпозиции принадлежность ее элементов к различным абстракциям данной предметной области. Вместо того чтобы разделять задачу на шаги типа «Получить

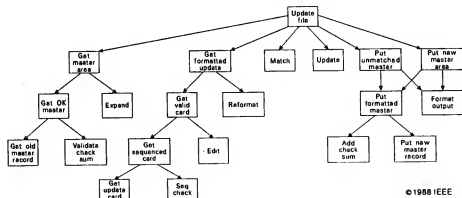


Рис. 1-2. Алгоритмическая декомпозиция.

изменения в отформатированном виде» и «Прибавить к контрольной сумме», мы идентифицировали объекты типа «Основной файл» и «Контрольная сумма», которые соответствуют словарю предметной области.

Хотя обе конструкции служат решению одной проблемы, они делают это разными способами. Во второй схеме мир представлен списком автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. «Получить изменения в отформатированном виде» больше не присутствует в качестве независимого алгоритма; вместо этого он существует в виде операций над объектом «Файл изменений». При вызове этой операции создается другой объект — «Изменение в строке». Каждый объект нашей системы обладает своим собственным поведением, моделирующим поведение реального объекта. С этой точки зрения объект является вполне определенной вещью, которая демонстрирует вполне определенное поведение. Объекты производят действия, и мы просим их сделать то-то и то-то, посылая им сообщения. Так как наша декомпозиция основана на объектах, а не на алгоритмах, мы называем ее объектной декомпозицией.

Алгоритмическая и объектно-ориентированная декомпозиции. Как правильно разделять сложную систему — по алгоритмам или по объектам? И по алгоритмам, и по объектам. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение факторам, либо вызывающим действия, либо являющимся объектами приложения этих действий¹⁾.

Однако мы не можем сконструировать сложную систему одновременно двумя способами, тем более что эти способы различны по сути. Мы можем начать разделение системы либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

Опыт показывает, что полезнее сначала применить объектный подход. Это поможет нам лучше понять структуру будущей программной системы. Выше мы уже применяли объектный подход при описании систем типа компьютеры, растения, галактики и общественные институты. В гл. 2 и 7 будут рассмотрены преимущества объектной декомпозиции. Ее использование дает возможность создавать программные системы меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств.

Объектно-ориентированные системы более открыты и легче поддаются модернизации, потому что конструкция таких систем базируется на устойчивых промежуточных формах. Кроме того, объектная декомпозиция уменьшает риск создания сверхсложных программных систем, так как она предполагает эволюционный путь развития системы на базе относительно небольших подсистем. Более того, объектная декомпозиция непосредственно определяет сложность программ, помогая нам принимать верные решения, когда вопрос касается разделения понятий в больших пространствах состояний.

¹⁾ Лонгдон замечает, что этот вопрос поднимался еще в древние времена. Как он утверждает: «Вздуингтон писал, что двойственность взглядов можно проследить вплоть до древних греков. Пассивный взгляд выдвигался Демокритом, который утверждал, что мир состоит из атомов. Он концентрирует внимание на вещах. С другой стороны, классический представитель активного взгляда на мир Гераклит уделял главное внимание процессу» [17].



Рис. 1-3. Объектно-ориентированная декомпозиция.

Преимущества объектно-ориентированных систем становятся видны при рассмотрении в гл. 8 — 12 ряда примеров прикладных программ, предназначенных для решения различных задач.

Абстракции

Выше мы говорили об экспериментах Миллера, в которых было установлено, что обычно человек может одновременно воспринять не более семи событий. Это число, по-видимому, не зависит от количества информации, содержащейся в событиях. Как замечает Миллер: «Размер нашей памяти накладывает жесткие ограничения на количество информации, которое мы можем воспринять, обработать и запомнить. Организуя поток входной информации одновременно по нескольким разным каналам и в виде последовательности отдельных событий, мы можем прорвать ... информационный затор» [18]. Он называл это декодированием, сейчас это называют разбиением или выделением абстракций.

Вулф описывает процесс декодирования информации в мозгу у человека следующим образом: «Люди обладают чрезвычайно эффективными механизмами обработки сложных сообщений, поступающих к ним через органы чувств. Они абстрагируются от них. Не в состоянии полностью воссоздать сложный объект, они просто игнорируют не слишком важные детали и, таким образом, имеют дело с обобщением, идеализированной моделью объекта» [19]. Например, изучая процесс фотосинтеза, мы концентрируем внимание на химических реакциях в определенных клетках листа и не обращаем внимание на остальные части — корни, стебли и т.д. И хотя мы при этом также ограничены количеством информации, которую можем воспринять в каждый отдельный момент, появляется возможность, используя абстрактные понятия, обрабатывать потоки сообщений, обладающие гораздо большей плотностью.

В гл. 2 понятие абстракции рассмотрено более детально.

Методы проектирования программных систем

Мы решили, что будет полезно, если мы разграничим понятия «метод» и «методология». Метод — это последовательный процесс создания ряда моделей, которые описывают вполне определенными средствами различные стороны разрабатываемой программной системы. Методология — это совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом. Эти механизмы необходимы по нескольким причинам. Во-первых, они упорядочивают процесс создания программных систем, являясь общими для всей группы разработчиков. Кроме того, позволяют менеджерам в процессе разработки оценить степень прогресса. Методы появились как ответ на растущую сложность программных систем. Раньше, например, очень трудно было написать большую программу, потому что возможности ЭВМ были ограничены. При разработке и развитии программных комплексов ограничения на них устанавливались, исходя из объема оперативной памяти, скорости считывания информации с вторичных носителей (ими служили магнитные барабаны) и быстродействия процессора, значение тактовой частоты которого измерялось сотнями микросекунд. В 60 — 70-е годы эффективность применения компьютеров резко возросла, так как цены на них стали падать, а возможности ЭВМ увеличились. В результате стало выгодно, да и необходимо создавать все больше прикладных программ повышенной сложности. В качестве основных инструментов создания программных продуктов начали применяться алгоритмические языки высокого уровня. Эти языки расширили возможности отдельных программистов и групп разработчиков, что в свою очередь привело к увеличению уровня сложности программных систем.

В 60—70-е годы было разработано много методов, помогающих справиться с растущей сложностью программ. Наибольшее распространение получило структурное проектирование по методу сверху вниз, или комбинированный метод. Он был непосредственно основан на топологии традиционных языков высокого уровня типа FORTRAN и COBOL. В этих языках основной базовой единицей является подпрограмма, и программа в целом принимает форму дерева, в котором одни подпрограммы в процессе работы вызывают другие подпрограммы. Структурное проектирование использует именно такой подход: алгоритмическая декомпозиция применяется для разбиения большой задачи на маленькие.

Начиная с 60—70-х годов стали появляться компьютеры еще больших, поистине громадных возможностей. Значение структурного подхода осталось прежним, но как замечает Стайн, «оказалось, что структурный подход не работает, если объем программы превышает приблизительно 100 000 строк» [20]. В последнее время появились десятки методов, в большинстве которых устранены очевидные недостатки структурного проектирования. Наиболее удачные методы были разработаны Питерсом [21], Яу и Цае [22], а также фирмой «Teledyne — Brown Engineering» [23]. Большинство этих методов представляют собой вариации на одну и ту же тему. Соммервилль, однако, предлагает разделить их на три основные группы [24]:

- * Метод структурного проектирования сверху вниз.
- * Метод организации потоков данных.
- * Объектно-ориентированное проектирование.

Примеры методов структурного проектирования приведены в работах Йордана и Коистантина [25], Маерса [26] и Пейджа-Джонсона [27]. Основы его изложены в работах Вирта [28, 29] и Даля, Динкстры и Хоара [30]; интересный вариант структурного подхода можно найти в работе Миллса, Лингера и Хевьера [31]. В каждом из этих подходов присутствует алгоритмическая декомпозиция. Следует отметить, что большинство существующих программ написано, по-видимому, в соответствии с одним из этих методов. Тем не менее структурный подход не позволяет выделять абстракции и обеспечивать защиту доступа к данным, не предоставляет он также достаточных средств для организации параллелизма. Структурный метод не может обеспечить создание предельно сложных систем, и он, как правило, неэффективен при использовании объектно-ориентированных языков программирования.

Метод организации потоков данных полнее всего описан в ранней работе Джексона [32, 33], а также Уорниера и Орра [34]. В этом методе структура программной системы строится как организация преобразований входных потоков в выходные. Метод организации потоков данных, как и структурный метод, с успехом применялся при решении ряда сложных задач, в частности, в системах информационного обеспечения, где существуют прямые связи между входными и выходными потоками системы и где не требуется уделять много внимания быстродействию.

Объектно-ориентированное проектирование (ООД) — это подход, основы которого изложены в данной книге. В основе ООД лежит представление о том, что программную систему необходимо проектировать как совокупность взаимодействующих друг с другом объектов, рассматривая каждый объект как экземпляр определенного класса, причем классы при этом образуют иерархию. Объектно-ориентированный подход отражает топологию новейших языков высокого уровня, таких, как Smalltalk, Object Pascal, C++, CLOS и Ada.

Иерархия

Другим способом, расширяющим возможности обработки потоков данных программной системой, является организация внутри системы иерархии классов и объектов. Структура объектов дает схему их взаимодействий друг с другом, которые осуществляются с помощью механизмов взаимодействий. Структура класса не менее важна — она определяет избыточность средств внутри системы. Зачем, например, изучать фотосинтез каждой клетки отдельного листа растения, когда достаточно изучить одну такую клетку, потому что скорее всего все остальные ведут себя подобным же образом. И хотя мы рассматриваем каждый объект определенного типа как отдельный, можно предположить, что его поведение будет похоже на поведение других объектов того же типа. Классифицируя объекты по группам родственных абстракций (например, типы клеток растений и клеток животных), мы четко разделяем общие и отличительные свойства разных объектов, что помогает нам затем создать их собственную сложную структуру [35].

Определить иерархии в сложной программной системе не всегда легко, так как это требует разработки моделей многих объектов, поведение каждого из которых может отличаться чрезвычайной сложностью. Однако

после их определения структура сложной системы и в свою очередь наше ее понимание сразу во многом проясняются. В гл. 3 в деталях рассматривается природа классов и иерархий объектов, а в гл. 4 описываются способы, облегчающие понимание их структуры.

1.4. ПРОЕКТИРОВАНИЕ СЛОЖНЫХ СИСТЕМ

Методология проектирования как наука и искусство

На практике каждая инженерная дисциплина — касается ли она строительства, механики, химии, электроники или программирования — включает элементы и науки, и искусства. Петроски по этому поводу утверждает следующее: «Понятие «разработка новых структур» предполагает и полет фантазии, и синтез опыта и знаний — все то, что необходимо художнику для реализации своего замысла на холсте или бумаге. И после того, как этот замысел созрел в голове инженера-художника, он обязательно должен быть проанализирован инженером-ученым со всей тщательностью, присущей настоящему ученому» [36].

Когда встает задача разработки совершенно новой системы, роль инженера-художника выдвигается на первый план. А ведь это довольно распространенный случай, особенно если мы имеем дело с системами, обладающими обратной связью, с системами управления и контроля, когда нам приходится писать программное обеспечение, требования к которому нестандартны, и для процессора, специально сконструированного для решения предложенной задачи. В других случаях, например при создании прикладных научных средств, инструментов для исследований в области искусственного интеллекта или для систем обработки информации, требования к системе могут быть хорошо и точно определены, но определены таким образом, что соответствующий им технический уровень разработки выходит за пределы существующих технологий. Нам, например, могут предложить создать систему, обладающую большим быстродействием, большей вместимостью или имеющей гораздо более мощные функциональные возможности по сравнению с уже существующими программными комплексами. Во всех этих случаях мы будем стараться использовать знакомые абстракции и механизмы («устойчивые промежуточные формы» в соответствии с терминологией, предложенной Саймоном) как основу будущей системы. При наличии большой библиотеки стандартных программных модулей инженер-программист должен просто по-новому скомпоновать эти модули таким способом, чтобы удовлетворить всем явным и неявным требованиям к системе, точно так же как художник или музыкант раздвигает пределы своих возможностей при создании новых произведений. Но так как подобных богатых библиотек практически не существует, инженер-программист может, к сожалению, обычно использовать лишь относительно небогатый список готовых модулей.

Цели проектирования

В любой инженерной дисциплине под проектированием обычно понимается некий строгий подход, с помощью которого мы ищем пути решения определенной проблемы, обеспечивая, таким образом, переход от требований к их исполнению. В контексте дисциплины инженерного программирования Мостоу определил цель проектирования создание — системы, которая :

- * «Удовлетворяет данным (возможно, неформальным) функциональным требованиям.
- * Имеет приемлемую цену.
- * Удовлетворяет явным и неявным требованиям по эксплуатационным качествам и ресурсопотреблению.
- * Удовлетворяет явным и неявным критериям дизайна.
- * Удовлетворяет требованиям к самому процессу разработки, таким, например, как его стоимость и продолжительность, а также не требует привлечения дополнительных инструментальных средств» [37].

Проектирование подразумевает учет противоречивых требований. Продуктами его являются модели, позволяющие нам понять структуру будущей системы, сбалансировать требования и наметить схему ее применения.

Важность построения модели. Моделирование широко распространено во всех инженерных дисциплинах. Причиной этому является то, что оно реализует принципы декомпозиции, абстракции и иерархии [38]. Каждая модель описывает определенную часть рассматриваемой системы, а мы в свою очередь строим новые модели на базе старых, в которых более ранее уже успели разобраться. Модели дают нам возможность исследовать недостатки системы в условиях, задаваемых нами самими. Мы оцениваем поведение каждой модели в обычных и необычных ситуациях, а затем проводим соответствующие доработки, если их поведение не чем-то не удовлетворяет.

Как мы уже сказали выше, чтобы понять во всех тонкостях поведение сложной системы, приходится использовать не одну модель. Например, проектируя компьютер, инженер-электронщик должен рассматривать электронную схему и знать при этом физическое расположение элементов на плате. Электронная схема формирует логическую картину компоновки системы, помогая инженеру разобраться в том, каким образом происходит взаимодействие между ее различными элементами. Схема платы представляет собой план физической реализации системы, ограниченный размером платы, потребляемой мощностью и типами имеющихся микросхем. С этой точки зрения инженер может независимо друг от друга оценивать такие параметры системы как температурное распределение и технологичность. Проектировщик платы может также рассматривать динамические и статические особенности разрабатываемой системы. Аналогично инженер-электрик может использовать диаграммы, иллюстрирующие статические связи между различными элементами, и временные диаграммы, отражающие поведение элементов во времени. Затем инженер может применить осциллограф или цифровой анализатор для проверки точности статических и динамических схем.

Элементы методов проектирования программного обеспечения. Ясно, что не существует такого универсального метода, который проведет инженера-программиста по пути от требований к сложной программной системе до их выполнения. Проектирование сложной программной системы отнюдь не сводится к следованию некоему набору рецептов. Скорее это постепенный и итеративный процесс. И тем не менее использование методов проектирования вносит в процесс разработки определенную организованность. Инженеры-программисты разработали десятки различных методов, которые мы можем классифицировать по трем категориям. Несмотря на различия, эти методы имеют что-то общее. Их, в частности, объединяет следующее:



Рис. 1-4. Модели объектно-ориентированного проектирования.

- * Условные обозначения Язык для описания каждой модели.
- * Процесс Правила упорядоченного проектирования модели.
- * Инструменты Средства, которые ускоряют процесс создания моделей, определяют законы их функционирования и помогают выявлять ошибки в процессе разработки.

Хороший метод проектирования базируется на солидной теоретической основе и при этом даст программисту известную степень свободы в выборе выразительных средств.

Модели объектно-ориентированного проектирования. Существует ли наилучший метод проектирования? Если и существует, то пока он никому не известен. Этот вопрос можно поставить следующим образом: как наилучшим способом разделить сложную систему на подсистемы? Еще раз напомним, что полезнее всего создавать такую модель системы, которая основывается на объектах, принадлежащих проблемной области, и сформирована как результат применения объектно-ориентированной декомпозиции.

Объектно-ориентированное проектирование — это метод, логически приводящий нас к декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем открытые для изменений программы, собранные из ограниченного числа универсальных блоков. Правильно разделив систему на относительно автономные небольшие подсистемы и по отдельности отладив каждую из них, мы в гораздо большей степени можем быть затем уверены, что наш конечный продукт будет свободен от ошибок. Таким образом, мы уменьшаем риск при разработке сложных программных систем.

Важность построения моделей при проектировании сложных систем диктует необходимость наличия нескольких типов моделей. Они представлены на рис. 1-4 и охватывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы. В гл. 5 подробно рассмотрен каждый из четырех типов моделей. В гл. 6 описан процесс объектно-ориентированного проектирования, представляющий собой последовательную цепь шагов по созданию и обеспечению развития данных моделей. В гл. 7 рассмотрены конкретные меры по управлению ходом работ с использованием объектно-ориентированного проектирования.

В этой главе мы привели доводы в пользу применения объектно-ориентированного проектирования для создания сложных структур при разработке программных систем. Кроме того, мы определили ряд преимуществ, достигаемых в результате применения такого подхода. Прежде чем мы начнем изучать сам процесс объектно-ориентированного проектирования, мы изучим те принципы, на которых он основан: выделение абстракций, ограничение доступа, модульность, иерархия, типирование, параллельность и устойчивость.

Выводы

- * Программам присуща сложность, которая нередко превосходит возможности человеческого разума.
- * Сложные структуры часто принимают форму иерархии; полезно моделировать и «типовую», и «структурную» иерархии сложной системы.
- * Сложные системы обычно создаются на основе устойчивых промежуточных форм.
- * Познавательные способности человека ограничены; мы можем раздвинуть их рамки, используя методы декомпозиции, выделения абстракций и создания иерархий.
- * Сложные системы можно исследовать, концентрируя основное внимание либо на объектах, либо на процессах; выгоднее рассматривать систему как упорядоченную совокупность объектов, которые в процессе взаимодействия друг с другом обеспечивают функционирование системы как единого целого.
- * Объектно-ориентированное проектирование — метод, использующий объектную декомпозицию; объектно-ориентированный подход имеет свою систему условных обозначений и предлагает богатый набор логических и физических моделей для проектирования систем высокой степени сложности.

Дополнительная литература

Вопросы, связанные с разработкой сложных программных систем, хорошо проработаны в классических работах Brooks [H, 1975, 1987], Glass [H, 1982], the Defense Science Board [H, 1987] и the Joint Service Task Force [H, 1982] могут дать дополнительную информацию по применению современного программного обеспечения. Работа Simon [A, 1982] — полезный справочник по структурам сложных систем; Courtois [A, 1985] применяет эти идеи в области программного обеспечения. Peter [I, 1986] и Petroski [I, 1985] исследуют сложные структуры применительно к общественным и физическим системам. В работе Flood and Carson [A, 1956] представлены теоретические исследования сложных структур с помощью теории систем. Доклад Miller [A, 1956] содержит эмпирические доказательства наличия ограничений на человеческие познавательные способности. Существует целый ряд прекрасных справочников по программированию. Ross, Goodenough and Irvine [H, 1980] и Zelkowitz [H, 1978] — два классических документа, содержащие все необходимые сведения о науке программирования. По этому же предмету существуют большие работы Jensen and Tones [H, 1979], Sommerville [H, 1985], Vick and Ramamoorthy [H, 1984], Wegner [H, 1980] и Pressman [H, 1987]. Другие статьи, относящиеся к созданию программного обеспечения, можно найти в работах Yourdon [H, 1979] и Freeman and Wasserman [H, 1983]. Gleick [I, 1987] предлагает читателю доступно написанное введение в науку хаоса.

Глава 2

Объектный подход

В основе объектно-ориентированного проектирования (ООД) лежит объектный подход. Основными принципами являются: абстрагирование, ограничение доступа, модульность, иерархичность, типизация, параллелизм и устойчивость. Эти принципы не новы, однако, именно в объектном подходе они объединены для решения общей задачи.

Объектно-ориентированное проектирование принципиально отличается от традиционных подходов структурного проектирования, так как подразумевает иной подход к процессу декомпозиции, а получаемый программный продукт по архитектуре в значительной степени выходит за рамки традиционных представлений. Отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования (ООП). К сожалению, единого понимания термина «объектно-ориентированное программирование» пока не достигнуто. Рентч высказал следующее предположение: «В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, которое занимало структурное программирование в 1970-х годах. Бесспорно, продукция всех фирм будет ориентирована на поддержку ООП. Каждый программист в той или иной степени будет использовать ООП. Но не все будут понимать причины этого положения» [1].

В этой главе сделана попытка выявить особенности ООД и отличия этого метода проектирования от других с учетом роли семи перечисленных выше элементов объектного подхода.

2.1. СТАНОВЛЕНИЕ ОБЪЕКТНОГО ПОДХОДА

Тенденции в методологии проектирования программных средств

Поколения языков программирования. Если проследить короткую, но пеструю историю развития методов программирования, можно выделить две основные тенденции:

- * Перемещение акцентов от программирования отдельных деталей к программированию более крупных компонент.
- * Развитие и совершенствование языков программирования высокого уровня.

Большинство современных коммерческих программных систем существенно сложнее и объемнее, чем их предшественники. Рост сложности обусловил проведение серьезных исследований в области методологии проектирования программных систем; в частности, были разработаны методы декомпозиции, абстрагирования и построения иерархии. Кроме того, были созданы более выразительные языки программирования. Возникла тенденция перехода от процедурных языков программирования (описывающих действия компьютера)

к декларативным языкам (описывающим ключевые абстракции проблемной области).

Вэгнер [2] следующим образом сгруппировал наиболее известные языки программирования высокого уровня в их поколениях:

• Первое поколение (1954—1958)

FORTRAN I

ALGOL-58

Flowmatic

IPL V

Математические формулы

Математические формулы

Математические формулы

Математические формулы

• Второе поколение (1959—1961)

FORTRAN II

ALGOL-60

COBOL

Lisp

Подпрограммы, раздельная компиляция

Блочная структура, типы данных

Описание данных, работа с файлами

Обработка списков, указатели

• Третье поколение (1962—1970)

PL/I

ALGOL-68

Pascal

Simula

Fortran+ALGOL+COBOL

Преемник ALGOL-60

Развитие ALGOL-60

Классы, абстрактные данные

• В 1970—1980-е годы возникло множество языков, из которых лишь немногие доказали свою жизнеспособность.

В каждом последующем поколении механизмы абстракции претерпевали изменения. Языки первого поколения ориентировались на научно-инженерные приложения, и словарь предметной области был исключительно математическим. Назначение таких языков, как FORTRAN I, состояло в упрощенном по сравнению с ассемблером или машинным кодом написании математических формул. Таким образом, первое поколение языков было шагом в направлении предметной области и отвлечением от особенностей компьютера. Во втором поколении языков основной тенденцией оказалось развитие алгоритмических абстракций. В это же время существенно выросла вычислительная мощность компьютеров и снизилась их стоимость, что позволило расширить область их применения, особенно в экономике. Главной задачей стала проблема ввода в машину инструкций: как считывать данные, сортировать их и выводить результаты на печать. Это был еще один шаг в направлении предметной области. В конце 60-х годов появление транзисторов, а затем интегральных схем резко снизило стоимость компьютеров, а их производительность росла почти экспоненциально. Возникла возможность решать все более крупные задачи, но это требовало умения обрабатывать разнородные данные. С возможностью обработки абстрактных данных связано появление языка ALGOL-60 и затем языка Pascal. Программисты получили возмож-

ность описания разнообразных видов данных (типов) и обработки их с помощью этих языков. Это стало еще одним шагом в направлении приближения к предметной области.

В 70-х годах были созданы тысячи различных языков и диалектов. Неадекватность более ранних языков задаче написания все более крупных программных систем стала очевидной, поэтому новые языки имели механизмы устранения этого препятствия. Лишь часть из новых языков смогла выжить (уже трудно найти литературу по языкам Fred, Chaos, Tranquil), однако многие их принципы нашли отражение в новых версиях более ранних языков. Таким образом, мы получили языки Ada (наследник ALGOL-68 и Pascal с элементами Simula, Alghard и CLU), CLOS (объединивший Lisp, LOOPS и Flavors) и C++ (возникший в результате слияния C и Simula). Наибольший интерес для дальнейшего изложения представляет класс языков, называемых объектными и объектно-ориентированными, которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции.

Архитектура языков программирования первого и второго поколения. Для пояснения сказанного следует рассмотреть каждое поколение языков с несколько иной точки зрения. На рис. 2-1 показана архитектура большинства языков первого поколения и первой стадии второго поколения. Такая архитектура отражает основные элементы конструкции языка программирования и их взаимодействие. Можно отметить, что для таких языков, как FORTRAN и COBOL, основным элементом конструкции является подпрограмма (в нотации COBOL — параграф). Программы, реализованные на таких языках, имеют относительно простую структуру, состоящую из области глобальных данных и подпрограмм. Стрелками на рисунке обозначено воздействие подпрограмм на различные данные. В процессе разработки программ имеется возможность логического разделения разнотипных данных, но механизмы языков практически не поддерживают такого разделения. Ошибка в какой-либо части программы может иметь далеко идущие последствия, так как область данных является общей для всех подпрограмм. В больших системах трудно гарантировать их целостность в процессе внесения изменений в какую-либо часть системы. В процессе эксплуатации уже через короткое время возникает путаница из-за большого количества перекрестных связей между подпрограммами, беспорядочности потоков управления, неопределенности данных, что снижает надежность системы и достоверность результатов.

Архитектура языков программирования второго и начальной стадии третьего поколения. Начиная с середины 60-х годов программы начали, наконец, приобретать статус существенного промежуточного звена между решаемой задачей и компьютером [3]. Шао отмечал: «Первая абстракция в программном продукте, названная процедурой, прямо вытекает из традиционного взгляда на программные средства. Подпрограммы возникли до 1950 г., но тогда не были оценены в качестве абстракции. Они представлялись в виде средств, упрощающих работу. Но очень скоро стало ясно, что подпрограммы являются абстрактным отражением функций программного продукта» [4].

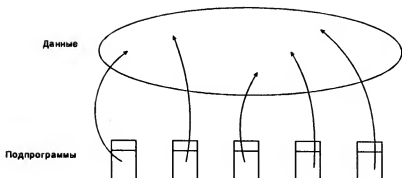


Рис. 2-1. Архитектура языков программирования первого и начальной стадии второго поколения.

Именно использование подпрограмм может служить механизмом абстрагирования, имеющим три существенных следствия. Во-первых, в языках появились механизмы, поддерживающие передачу параметров. Во-вторых, были заложены основания структурного программирования, отразившиеся в создании механизмов вложения подпрограмм, включая ограничения области действия (видимости) компонентов. В-третьих, возникли методы структурного проектирования, обеспечивающие создание больших систем на основе подпрограмм. Понятно, что изменилась и архитектура языков программирования (рис. 2-2). Такая архитектура разрешила ряд противоречий, возникших в предыдущем периоде, в частности усилила управление алгоритмическими абстракциями, но не смогла решить задачу обеспечения программирования высокого уровня и использования многообразия данных.

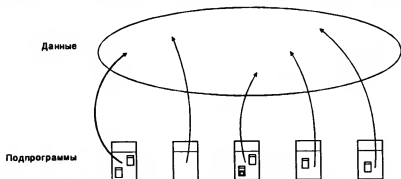


Рис. 2-2. Архитектура языков программирования второго и начальной стадии третьего поколения.

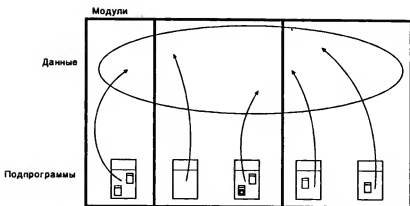


Рис. 2-3. Архитектура языков программирования третьего поколения (завершающая стадия).

Архитектура языков программирования третьего поколения (завершающая стадия). Начиная с FORTRAN II и позднее для решения задач программирования на более высоком уровне стали возникать новые существенные механизмы структурирования. Создание больших проектов вызвало необходимость организации коллективной работы программистов, разрабатывающих параллельно отдельные части общей системы.

Это привело к реализации механизма раздельной компиляции модулей, которые были чем-то большим, чем случайный набор данных и подпрограмм (рис. 2-3). Позднее модули стали важным механизмом абстракции, являясь вначале просто группой логически связанных подпрограмм. Для большинства языков этого поколения, использующих модульную структуру, слабо определены правила построения межмодульного интерфейса. Программист может предположить, что его подпрограмма будет вызываться с тремя параметрами: действительным числом, массивом из десяти элементов и целым числом, обозначающим булевский флаг управления. Вызов этой подпрограммы в другом модуле может сопровождаться иными параметрами: целым числом, массивом из пяти элементов и отрицательным числом. Поскольку средства поддержки абстракции данных и строгого типирования в таких языках недостаточны, ошибка может быть выявлена только во время выполнения такой программы.

Архитектура языков объектного и объектно-ориентированного программирования. Значение абстрактных типов данных в разрешении проблемы сложности систем с очевидностью установлена Шапкарном: «Суть абстрагирования, достигаемого посредством использования процедур, достаточна для описания абстрактных действий, но недостаточна для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуа-

циях сложность объектов, являющихся предметом управления, составляет основную часть сложности всей задачи» [5]. Из этого вытекают два важных следствия. Во-первых, возникают методы проектирования управляемыми данными, которые вносят порядок в обработку абстрактных данных алгоритмическими языками. Во-вторых, появляется теория типирования, которая воплощается в языках типа Pascal.

Естественным завершением реализации этих идей, начавшейся с языка Simula и развитой в последующих языках в 1970—1980-е годы, стало появление таких языков, как Smalltalk, Object Pascal, C++, CLOS и Ada. Именно эти языки получили название объектных или объектно-ориентированных. На рис. 2-4 приведена архитектура таких языков применительно к задачам малой и средней степени сложности. Основным элементом конструкции в указанных языках служит модуль, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения. Определим это следующим образом: «Если представить процедуры и функции в виде глаголов, а данные в виде имен существительных, то процедурные программы строятся из глаголов, а объектно-ориентированные из имен существительных» [6]. По этой же причине структура программ малой и средней сложности при объектно-ориентированном подходе представляется графом, а не деревом, как в случае алгоритмических языков. Кроме того, сокращена или отсутствует область глобальных данных. Данные и действия организуются теперь таким образом, что основой конструкции становятся классы и объекты, а не алгоритмы. В настоящее время идет процесс, направленный на решение задач большой и глобальной степени сложности. Для достаточно больших систем классы, объекты и модули обнаруживают недостаточный уровень декомпозиции.

К счастью, объектный подход может быть осуществлен на более высоких уровнях абстракций. Группы абстракций в больших системах могут представляться в виде многослойной структуры. На каждом уровне можно выделить группы объектов, тесно взаимодействующих для решения задачи более высокого уровня абстракции. Внутри каждой группы мы неизбежно найдем такое же множество взаимодействующих абстракций (рис. 2-5). Это соответствует подходу к сложным системам, изложенному в гл. 1.

Основные положения объектного подхода

Методы структурного проектирования имели своей целью упростить процесс разработки сложных систем на основе алгоритмического подхода. Методы объектно-ориентированного проектирования созданы в свою очередь для помощи разработчикам, использующим мощные выразительные средства объектного и объектно-ориентированного программирования, основанного на описании классов и объектов. В принципах объектного подхода нашли свое отражение и множество других факторов, кроме объектно-ориентированное программирование (ООП). Ниже показано, что объектный подход должен быть обобщенным подходом не только в программировании, но также в проектировании интерфейса пользователя, баз данных, баз знаний и даже компьютерной архитектуры. Смысл такого широкого подхода состоит в том, что он позволяет применить объектную ориентацию для решения всего круга проблем, связанных со сложными системами.

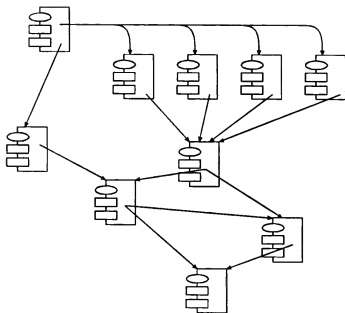


Рис. 2-4. Архитектура программных систем малой и средней сложности, использующих объектные и объектно-ориентированные языки программирования.

ООД отражает эволюционный процесс в проектировании, а не революционный; новая методология не является резким отходом от прежних методов, а строится с учетом предшествующего опыта. К сожалению, большинство программистов в настоящее время используют формально или неформально методы структурного проектирования. Разумеется, многие хорошие проектировщики создали и продолжают совершенствовать большое количество программных систем на основе этой методологии. Однако трудно преодолеть ограничения, связанные с уровнем сложности таких систем, не прибегая к объектно-ориентированной декомпозиции. Более того, если мы попытаемся использовать такие языки, как C++ или Ada, в качестве традиционных алгоритмических, мы не только потеряем их внутрисистемный потенциал, скорее всего результат будет хуже, чем при использовании обычных языков C и Pascal. Дать мощную электродрель плотнику, который не слышал об электричестве, значит использовать ее в качестве молотка. Но, прежде чем дрель превратится в негодный молоток, будет испорчено несколько гвоздей и разбиты пальцы.

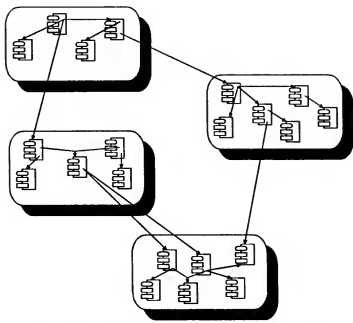


Рис. 2-5. Архитектура программных систем большой сложности на основе объектных и объектно-ориентированных языков программирования.

ООР, ООД и ООА

От множества своих предшественников объектный подход, к сожалению, унаследовал запутанную терминологию. Программирующие на Smalltalk пользуются термином «метод», на C++ — термином «фактическая функция-элемент», а в случае CLOS — «обобщенная функция». В Object Pascal используется термин «приведение типов», а в языке Ada та же вещь называется «преобразованием типов». Чтобы избежать путаницы, следует определить более точно понятие объектной ориентации. Определения наиболее употребляемых терминов и понятий вы можете найти в словаре в конце книги.

Термин «объектно-ориентированный», по мнению Бхаскара используется с легкой небрежностью теми, кто почтительно говорит «материнство», «кусочек яблока» и «структурное программирование» [7]. Можно согласиться, что понятие объекта является центральным во всем, что относится к объектно-ориентированной методологии. В гл. 1 мы определили объект как осязаемую сущность, которая четко проявляет свое поведение. Стефник и Бобров определяют объекты как «нечто, объединяющее свойства процедур и данных в процессе выполнения вычислений и сохраняющее локальное состояние» [8]. Такое определение объекта допустимо, но главным в понятии объекта является все же объединение идей абстрагирования данных и алгоритмов. Джонс

уточняет это понятие следующим образом: «В объектном подходе акцент переносится на конкретные характеристики физической и абстрактной системы, являющейся предметом программного моделирования... Объекты обладают целостностью, которую не следует нарушать. Объект может только менять состояние, поведение, управляться или становиться в определенное отношение к другим объектам. Объект обладает неизменными качествами, но может изменять свое состояние. Например, подъемник характеризуется тем, что может двигаться вверх и вниз, оставаясь в пределах своих направляющих ... Любая модель должна учитывать эти свойства подъемника, так как они составляют его назначение» [32].

Объектно-ориентированное программирование. Объектно-ориентированное программирование — это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследуемости.

В данном определении можно выделить три части: 1) ООР использует в качестве элементов конструкции объекты, а не алгоритмы (нерархия по составу была определена в гл. 1); 2) каждый объект является реализацией какого-либо определенного класса; 3) классы организованы иерархически (нерархия по номенклатуре обсуждена в гл. 1). Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование не основанное на иерархических отношениях, не относится к ООР, а называется программированием на основе абстрактных типов данных.

Исходя из этого, ясно, что не все языки программирования являются объектно-ориентированными. Страустрап определил так: «если термин «объектно-ориентированный язык» что-то означает, то он должен означать язык, имеющий все необходимые средства для обеспечения объектно-ориентированного стиля программирования ... Обеспечение такого стиля в свою очередь означает наличие соглашений по реализации стиля программирования. Если написание программ в стиле ООР требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям ООР» [33]. Теоретически возможна имитация объектно-ориентированного программирования на обычных языках, таких, как Pascal и даже COBOL или ассемблер, но это крайне затруднительно. Карделли и Вагнер утверждают: «язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

- * Имеется поддержка объектов в виде абстракции данных имеющих интерфейсную часть в виде поименованных операций и защищенную область локальных данных.
- * Объекты относятся к соответствующим типам (классам).
- * Типы (классы) могут наследовать атрибуты от супертипов (суперклассов)» [34].

Основные положения объектного подхода

Юиизава и Токоро свидетельствуют, что «термин «объект» появился практически независимо в различных областях, связанных с компьютеризацией, и почти одновременно в начале 70-х годов для обозначения того, что может иметь различные проявления, оставаясь целостным. Это требовалось для того, чтобы уменьшить сложность программных систем, обозначая объектами компоненты системы или фрагменты представляемых знаний» [9]. По мнению Леви, объектно-ориентированный подход был связан со следующими событиями [10]:

- «Прогресс в области архитектуры ЭВМ, включая системную и аппаратную поддержку.
- Развитие языков программирования, таких, как Simula, Smalltalk, CLU, Ada.
- Развитие методологии программирования, включая принципы модульности и защиты информации».

К этому еще следует добавить три момента, оказавшие влияние на становление объектного подхода:

- Развитие баз данных.
- Исследования в области искусственного интеллекта.
- Достижения философии и теории познания.

Понятие «объект» впервые было использовано более 20 лет назад в технических средствах при создании архитектур, основанных на описаниях и реализациях [11]. В этих работах делались попытки отойти от традиционной архитектуры Неймана и преодолеть барьер между высоким уровнем программных абстракций и низким уровнем абстрагирования на уровне ЭВМ [12]. При этом были созданы более качественные средства: лучшее выявление ошибок, большая эффективность реализации программ, сокращен набор команд, упрощена компиляция, снижены объемы требуемой памяти. Ряд компьютеров имеет объектно-ориентированную архитектуру: Burroughs 5000, Plessey 250, Cambridge CAP [13], SWARD [14], Intel 432 [15], Caltech's COM [16], IBM System/38 [17], Rational R1000, BiiN 40 и 60.

С объектно-ориентированной архитектурой тесно связаны объектно-ориентированные операционные системы (ОС). Дейкстра, работая над мультипрограммной системой THE, впервые ввел понятие структурированной машины состояний в качестве средства реализации систем [18]. Среди первых объектно-ориентированных ОС следует отметить: Plessey/System 250 (для мультипроцессора Plessey 250), Hydra (для CMU c.mmp), CALTSS (для CDC 6400), CAP (для Cambridge CAP), UCLA Secure Unix (для PDP 11/45 и 11/70), StarOS (для CMU Cm*), Medusa (также для CMU Cm*) и iMAX (для Intel 432) [19].

Наиболее значительный вклад в объектный подход внесли объектные и объектно-ориентированными языками программирования. Впервые понятия классов и объектов введены в языке Simula 67. Система Flex и

последовавшие за ней диалекты Smalltalk-72, -74, -76 и, наконец, -80, взяв за основу методы Simula, довели их до логического завершения, выполняя все действия на основе классов. В 1970-х годах создан ряд языков, реализующих идеи абстрактных данных: Alphard, CLU, Euclid, Gypsy, Mesa и Modula. Затем методы, используемые в языках Simula и Smalltalk, были использованы в традиционных языках высокого уровня. Внесение объектно-ориентированного подхода в С привело к возникновению языков C++ и Objective C. На основе языка Pascal возникли Object Pascal, Eiffel и Ada. Появились диалекты LISP, такие, как Flavors, LOOPS и CLOS (common LISP Object System), с возможностями языков Simula и Smalltalk. Более подробно особенности этих языков изложены в приложении. Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра. Позднее Парнас ввел идею защиты информации [20], а в 70-х годах ряд исследователей разработали механизмы абстрактных типов данных [21-23]. Хоар дополнил эти подходы теорией типов и подклассов [24].

Развивавшиеся достаточно независимо технологии построения баз данных также оказали влияние на объектный подход [25] в первую очередь благодаря идеям отношений сущности (ER) в моделировании баз данных [26]. В моделях ER, впервые изложенных Ченом [27], задачи представляются в терминах сущностей, их атрибутов и взаимоотношений. Разработчики способов представления данных в области искусственного интеллекта также внесли свой вклад в понимание объектно-ориентированных абстракций. В 1975 г. Мински выдвинул теорию фреймов для представления реальных объектов в системах распознавания образов и естественных языков [28]. Фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах. Объектный подход известен еще издавна. Грекам принадлежит идея о том, что мир можно рассматривать как в терминах объектов, так и событий. А в 17 в. Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир [29]. В 20 в. эту тему развивала Рауд в своей теории познания [30]. Позднее Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различных мыслящих агентов [31]. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека.

Поддержка наследуемости в таких языках означает возможность установления отношений вида «разновидность» между типами, например красная роза — разновидность цветов, а цветок — разновидность растений. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным. Карделли и Вагнер называли такие языки объективными, но не объектно-ориентированными. Исходя из этого, объектно-ориентированными языками являются Smalltalk, Object Pascal, C++ и CLOS, а Ada — объектный язык. Но, поскольку объекты и классы являются элементами обеих групп языков, методы ООД желательно использовать и в тех и в других.

Объектно-ориентированное проектирование. Методы программирования прежде всего подразумевают правильное и эффективное использование механизмов языков программирования. Методы проектирования, напротив, основное внимание направляют на правильное и эффективное структурирование сложных систем.

Определим объектно-ориентированное проектирование следующим образом:

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления как логической и физической, так статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части: 1) OOD ведет к объектно-ориентированной декомпозиции; 2) используется многообразие приемов представления моделей, отражающих логическую (структуры классов и объектов) и физическую (архитектура моделей и процессов) структуру системы.

Именно поддержка объектно-ориентированной декомпозиции отличает OOD от структурного проектирования; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором в виде алгоритмического обозначения методов, связанных с объектно-ориентированной декомпозицией.

Объектно-ориентированный анализ. На объектный подход оказали влияние предыдущие этапы развития программных средств. Традиционные приемы структурного анализа, известные по работам Де Марко [35], Йордана [36], Гае и Сарсона [37], с уточнениями для режимов реального времени Варда и Меллора [38], Хатли и Пирбой [39] основаны на потоках данных в системе.

Объектно-ориентированный анализ (ООА) направлен на создание моделей, более близких к реальности, с использованием объектно-ориентированного подхода; это методология, при которой требования формируются на основе понятий классов и объектов, составляющих словарь предметной области.

На результатах ООА формируются модели, на которых основывается OOD; OOD в свою очередь создаст основу для окончательной реализации системы с использованием методологии OOP.

2.2. КОМПОНЕНТЫ ОБЪЕКТНОГО ПОДХОДА

Способы программирования

Джекиис и Глазгов считают, что «большинство программистов используют в работе один язык программирования и один стиль. Приемы и способы программирования определяются используемым языком. Часто в стороне остаются альтернативные подходы к цели, а следовательно, не используются оптимальные решения в выборе стиля, соответствующего решаемой зада-

че» [40]. Бобров и Стефик так определили понятие стиля программирования: «Это способ построения программ, основанный на определенных принципах программирования и выборе языка, с целью добиться ясности понимания программы» [41]. Эти же авторы выявили пять основных разновидностей стиля программирования, которые перечислены ниже вместе с присущими им видам абстракций:

- | | |
|----------------------------------|--|
| * Процедурно-ориентированный | Алгоритмы |
| * Объектно-ориентированный | Классы и объекты |
| * Логически-ориентированный | Цели, наиболее часто выраженные в исчислениях предикатов |
| * Ориентированный на правила | Правила «если...то» |
| * Ориентированный на ограничения | Инвариантные соотношения |

Невозможно назвать какой-либо стиль программирования наилучшим во всех областях практического применения. Например, для проектирования баз знаний более пригоден стиль, ориентированный на правила. Рассматриваемый нами объектно-ориентированный стиль является наиболее приемлемым для широкого круга задач, связанных с большими промышленными системами, в которых основной проблемой является сложность.

Каждый стиль программирования имеет свою концептуальную основу, требует различного подхода к решаемой задаче. Для объектно-ориентированного стиля концептуальная основа состоит в объектном подходе. Этому подходу соответствуют четыре главных элемента:

- * Абстрагирование.
- * Ограничение доступа.
- * Модульность.
- * Иерархия.

Эти элементы являются главными в том смысле, что без любого из них подход не будет объектно-ориентированным. Кроме главных имеется еще три дополнительных элемента:

- * Типизация.
- * Параллелизм.
- * Устойчивость.

Эти элементы являются полезными, но не обязательными в объектном подходе. Отсутствие соответствующей концептуальной основы приведет к тому, что программы, написанные на языках Smalltalk, Object Pascal, C++, CLOS и Ada, будут мало отличаться от программ на FORTRAN, PASCAL или C. Выразительная способность этих объектных или объектно-ориентированных языков будет либо потеряна, либо искажена. Но еще более существенно, что при этом будет мало шансов «справиться» со сложностью решаемых задач.

Абстрагирование

Роль абстрагирования. Абстрагирование является одним из главных способов, используемых для решения сложных задач. Хоаре предположил, что «абстрагирование заключается в нахождении сходств между определенной со-

вокупностью объектов, ситуаций или процессов, имеющих место в реальности, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющих отличий между этими объектами» [42]. Шоу определил это понятие так: «Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, при которой подчеркиваются существенные для рассмотрения и использования детали и опускаются те, которые на данный момент несущественны или отвлекают внимание» [43]. Берзинс, Грей и Науман добавили к этому следующее: «Абстракцией является только такая идея, которая может быть изложена, понята и проанализирована независимо от механизма ее реализации» [44]. Суммируя все изложенное, получим следующее определение абстракции:

Абстракция — это такие существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от деталей их осуществления. Абельсон и Суссман назвали такое разделение (поведение от осуществления) барьером абстракции [45], который основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения [46]. Полезным является еще один дополнительный принцип, называемый принципом наименьшей выразительности, по которому абстракция должна охватывать лишь самую суть объекта, не больше, но и не меньше. Выбор достаточного множества абстракций для заданной предметной области является главной проблемой объектно-ориентированного проектирования. Гл. 4 целиком посвящена этой теме. По мнению Сейдсвитца и Старка, «существует целый спектр абстракций, начиная с объектов, которые приблизительно соответствуют сущности предметной области, кончая объектами, не имеющими реальных аналогий в жизни» [47]:

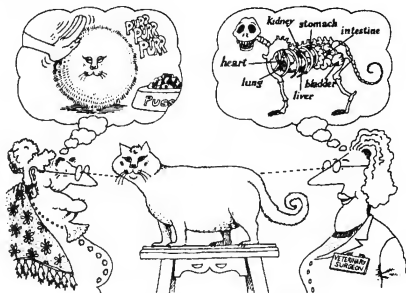
- * Абстракция сущности объекта Объект представляет собой модель существенных сторон предметной области
- * Абстракция поведения Объект состоит из обобщенного множества операций, каждая из которых выполняет определенную функцию
- * Абстрагирование в виде виртуальной машины Объект объединяет группы операций виртуальной машины, которые используются либо для управления объектом, либо соответствуют функциям нижнего уровня
- * Произвольная абстракция Объект включает в себя набор независимых по отношению друг к другу операций

Наиболее интересны для нас абстракции сущности объектов, так как они соответствуют словарю предметной области. Объект, использующий ресурсы других объектов, называется *клиентом*. Описание поведения объекта включает описание операций, которые могут выполняться над ним, и операций, которые сам объект выполняет над другими объектами. Такой подход концентрирует внимание на внешних особенностях объекта. Полный набор операций, которые объект может осуществлять над другим объектом, называется *протоколом*. Протокол отражает все действия, которым объект может

подвергаться сам и которыми может оказывать влияние на другие объекты, определяя тем самым полностью внешнее поведение абстракции со статической и динамической точек зрения.

Заметим, что понятия *операция*, *метод* и *функция* — элемент (operation, method, member function) происходят от различных традиций программирования (Ada, Smalltalk и C++ соответственно). Фактически они обозначают одно и то же и в дальнейшем будут использоваться эквиваленты.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, объект-файл требует определенного объема памяти, имеет имя и содержание. Эти атрибуты являются статическими. Конкретные значения каждого из перечисленных свойств являются динамическими, изменяющимися в процессе использования объекта: файл может изменять свои размеры, имя и содержимое. В процедурном стиле программирования изменения динамических характеристик объектов составляют суть программ. Любые события связаны с вызовом подпрограмм и с выполнением операторов. Стил программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые в свою очередь возбуждают другие правила и так далее. Объектно-ориентированный стиль программирования связан с воздействием на объекты (в языке Smalltalk объектам передают сообщения). Путем воздействия на объект вызывается определенная реакция этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия составляют характер поведения этого объекта.



Абстрагирование позволяет сосредоточить внимание на существенных характеристиках объекта с точки зрения наблюдателя.

Примеры абстракций. Для иллюстрации сказанного выше приведем несколько примеров. В данном случае мы сконцентрируем внимание не столько на том, как формируются абстракции для конкретной задачи (это подробно рассмотрено в гл. 4), сколько на способе выражения абстракций.

В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы. Управление режимом работы такого хозяйства очень серьезная и «токая» проблема, зависящая как от вида выращиваемых культур, так и от стадии выращивания. При этом нужно контролировать целый ряд таких факторов, как температура, влажность, освещение, кислотность (показатель pH) и концентрация питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Попросту говоря, цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения плана-графика выращивания хорошего урожая.

Одной из ключевых абстракций в такой задаче является датчик. Известно несколько разновидностей датчиков. Все важные воздействующие факторы должны быть измерены. Для этого существуют датчики температуры, влажности, pH, освещения и концентрации питательных веществ. С внешней точки зрения датчик температуры — это объект, который способен измерять температуру в определенном месте. Что такое температура? Это числовой параметр, имеющий ограниченный диапазон значений и определенную точность, означающий число градусов по Фаренгейту, Цельсию или Кельвину в зависимости от условий конкретной задачи. Что такое местоположение датчика? Это определенное указанное место в теплице, в котором нам необходимо знать температуру; таких мест, вероятно, несколько. Для датчика температуры существенно не столько само местоположение, сколько тот факт, что данный датчик расположен именно в данном месте и что отличает его от других датчиков. Теперь можно задать вопрос о том, какие действия можно выполнять по отношению к датчику? В нашем случае пользователь может калибровать датчик и получать от него значение текущей температуры.

Для демонстрации проектных решений ниже приводятся примеры на языке Ada. Читатели недостаточно знакомые с этим языком, а также желающие уточнить свои знания по другим объектным и объектно-ориентированным языкам, упоминаемым в этой книге, могут найти их краткие описания с примерами в приложении. На языке Ada абстрактное описание датчика температуры воздуха будет отражено следующей спецификацией:

```
package Temperature_Sensors is
```

```
  type Temperature is delta 0.01 range -10.0..150.0;
  — temperature in degrees Fahrenheit
```

```
  type location is range 0..63;
  — a number denoting the location of a sensor
  type Air_Temperature_Sensor is limited private;
  — the air temperature sensor class
```

```

procedure Initialize (The_Sensor : in Air_Temperature_Sensor;
                     Its_Location : in Location );

procedure Calibrate (The_Sensor : in out Air_Temperature_Sensor;
                    Actual_Temperature : in Temperature);

function Current_Temperature (The_Sensor : in Air_Temperature_Sensor )
    return Temperature;

private
...
end Temperature_Sensor;

```

Этот модуль определяет (экспортирует) три типа данных: Temperature, Location и Air_Temperature_Sensor. Тип Temperature представляется числом с фиксированной точкой, означающим температуру по Фаренгейту. Тип Location обозначает вариант размещения датчика. Наконец, тип Air_Temperature_Sensor составляет собственно абстракцию датчика, представление которой выполнено защищенным.

Поскольку приведенное описание является описанием класса, прежде чем выполнить какие-либо действия, необходимо создать экземпляр объекта (реализацию) данного класса, например, следующим образом:

```

with Temperature_Sensor;
use Temperature_Sensor;

...
Greenhouse_1_Temperature_Sensor : Air_Temperature_Sensor;
Greenhouse_2_Temperature_Sensor : Air_Temperature_Sensor;
The_Temperature : Temperature;

begin
    Initialize(Greenhouse_1_Temperature_Sensor, Its_Location => 1);
    Initialize(Greenhouse_2_Temperature_Sensor, Its_Location => 2);
    The_Temperature := Current_Temperature (Greenhouse_1_Temperature_Sensor);
    ...
end;

```

Описанная абстракция является пассивной; другие объекты могут воздействовать на этот объект (датчик температуры), чтобы получить его значение. Описание объекта температурного датчика можно сделать и активным, чтобы он воздействовал на другие объекты в случае, если измеряемая температура изменится на некоторое число градусов. Такая абстракция будет похожа на предыдущую, но ее интерфейс становится двухсторонним, и будет выглядеть следующим образом:

```

package Temperature_Sensors is
    type Temperature is delta 0.01 range -10.0..150.0;
    -- temperature in degrees Fahrenheit

    type Location is range 0..63;
    -- a number denoting the location of a sensor

    generic
        with procedure Temperature_Has_Changed (The_Location : in Location;
                                                New_Temperature : in Temperature);
        with procedure Temperature_Alarm (The_Location : in Location;
                                         New_Temperature : in Temperature);
    package Temperature_Sensors is

```

```

type Air_Temperature_Sensor is limited private;
-- the air temperature sensor class

procedure Initialize (The_Sensor           : in Air_Temperature_Sensor;
                     Its_Location          : in Location;
                     Lower_Alarm_Limit     : in Temperature;
                     Upper_Alarm_Limit     : in Temperature);

procedure Calibrate (The_Sensor           : in out Air_Temperature_Sensor;
                    Actual_Temperature    : in Temperature);

private
...
end Air_Temperature_Sensors;

end Temperature_Sensors;

```

Такое описание сложнее первого, но более полно соответствует описываемой абстракции. Над указанным объектом можно выполнять только две операции: инициализацию и калибровку. В процессе существования каждый из объектов типа датчика температуры имеет возможность формировать сообщения об изменении температуры и аварийной ситуации для передачи в другие объекты. На данном примере, таким образом, показано использование языка Ada для описания действий над объектом и выполнения воздействий самого объекта на другие объекты.

Рассмотрим другой пример абстракции, на языке C++. В тепличном хозяйстве с целью обеспечения наибольшего урожая необходим план выращивания, учитывающий влияние факторов температуры, освещения, вносимых питательных веществ и ряда других на выращивание культур. Поскольку такой план является частью словаря предметной области, вполне оправдана его реализация в виде абстракции.

Для каждой выращиваемой культуры существует отдельный такой план, но общая форма планов во всех случаях одинакова. Основу плана выращивания составляет таблица, отражающая во времени перечень необходимых действий. Например, для некоторой культуры на 15-е сутки ее роста план предусматривает поддержание в течении 16 ч температуры 78° F, из них 15 ч с освещением, а затем понижение температуры до 65° F на остальное время суток. Кроме того, может потребоваться внесение определенного вещества для поддержания заданной кислотности в почве (показатель pH).

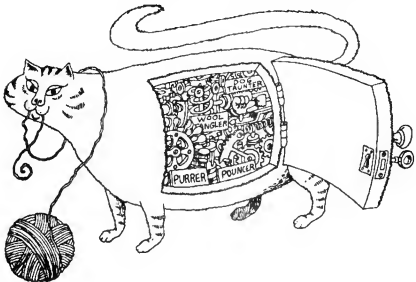
С точки зрения интерфейса объекта-плана необходимо обеспечить возможность задания деталей плана, изменять план и осуществлять его выполнение. Например, возможно наличие объекта, обеспечивающего интерфейс человек-компьютер и ручное изменение плана. Это означает внесение изменений в объект-план с целью корректировки отдельных деталей. Кроме того, должен существовать объект-исполнитель плана, имеющий возможность читать данные о плане. Как видно из дальнейшего описания, ни один объект не обособлен, а все они взаимодействуют для обеспечения общей цели. Исходя из такого подхода, определяются и границы каждого объекта-абстракции, и протоколы их связи.

Реализация плана выращивания на языке C++ будет выглядеть следующим образом:

```
typedef int day;
typedef int hour;
typedef float temperature;
typedef float ph;
typedef float concentration;
enum boolean(OFF, ON);

class GrowingPlan {
...
public:
    GrowingPlan ();
    GrowingPlan (const GrowingPlan&);
    virtual ~GrowingPlan ();

    virtual void clearThePlan ();
    virtual void establish      (day      theDay,
                                hour      theHour,
                                temperature theTemperature,
                                boolean    lightsOn,
                                ph         thePh,
                                concentration theNutrientConcentration);
    virtual temperature desiredTemperature (day theDay, hour theHour) const;
    virtual boolean    lighthStatus       (day theDay, hour theHour) const;
    virtual ph         desiredPh          (day theDay, hour theHour) const;
    virtual concentration desiredNutrients (day theDay, hour theHour) const;
};
```



Ограничение доступа позволяет скрыть детали устройства объекта.

Поясним использование в этом примере определения типов. Стиль программирования требует явного объявления всех типов, составляющих словарь предметной области, если это не противоречит другим обстоятельствам. Интерфейс объекта-плана выращивания намеренно лишен обособленных составляющих, чтобы сосредоточить внимание на поведении объекта, а не на особенностях его строения. В языке C++ обособленными являются все элементы описания, если явно не определено другое.

В общедоступную часть описания (*public*) вынесены *конструктор* и *деструктор объекта* (определяющие процедуры порождения и уничтожения данного объекта), две *процедуры модификации* (очистка всего плана и определение элементов плана) и четыре *селектора-определителя* состояния (по одному для каждого элемента состояния плана выращивания) плана на данный момент. Стиль программирования подразумевает определение всех функций-элементов объекта в качестве фактических (*virtual*), чтобы сохранить возможность их переопределения в подклассах (если по другим причинам не требуется иного).

Так же как в случае примера на языке Ada, класс план-выращивания является абстракцией, а не объектом, пригодным для использования. Поэтому требуется в каком-то определенном месте программы создать экземпляр объекта данного класса и использовать его затем для соответствующих манипуляций.

Ограничение доступа

Понятие ограничения доступа. Созданию абстракции какого-либо объекта должны предшествовать определенные решения о способе ее реализации. Выбранный способ реализации должен быть скрыт и защищен для большинства объектов-пользователей (обращающихся к данной абстракции). Как справедливо отметил Ингалс, «никакая часть сложной системы не должна находиться в зависимости от подробностей внутреннего устройства других частей системы» [48]. Ограничение доступа «позволяет вносить в программу изменения, сохраняя ее надежность и минимизируя затраты на этот процесс» [49].

Абстрагирование и ограничение доступа являются взаимодополняющими операциями: абстрагирование фокусирует внимание на внешних особенностях объекта, а *ограничение доступа* — или иначе *защита информации* — не позволяет объектам-пользователям различать внутреннее устройство объекта. Ограничение доступа, таким образом, определяет явные барьеры между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как корни растения или митохондрии клеток. Аналогичным образом при проектировании баз данных программисты не обращают внимание на физический смысл данных, а сосредоточиваются на схеме, отражающей логическое строение данных [50].

В обоих случаях объекты верхнего уровня абстракции не связаны прямо с подробностями их реализации на низком уровне. Лисков считает что «для «работы» абстракции, доступ к ее внутренней структуре должен быть ограничен» [51]. Практически это означает наличие двух частей в описании класса: интерфейса и реализации. *Интерфейс* отражает внешнее проявление объекта, создавая абстракцию поведения всех объектов данного класса. *Внутренняя реализация* описывает механизмы достижения желаемого поведения

объекта. Принцип такого различия интерфейса и реализации соответствует разделению по сути: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. Бриттон и Парнас называли такие подробности «тайнами» абстракции [52].

Итак, понятие ограничения доступа можно определить следующим образом:

Ограничение доступа — это процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта как целого.

На практике осуществляется защита как структуры объекта, так и реализации его методов.

Примеры использования ограничения доступа. Для иллюстрации практического использования ограничения доступа вернемся к примеру тепличного хозяйства. Одной из ключевых абстракций данной предметной области является абстракция обогревателя, поддерживающего заданную температуру в помещении. Обогреватель является абстракцией низкого уровня, поэтому можно предположить достаточным выполнение над этим объектом всего трех действий: включения, выключения и проверки работоспособности. В соответствии с принятым стилем описания должны быть также определены операции создания и ликвидации объектов — конструктор и деструктор. Реальная система может содержать множество обогревателей, поэтому для привязки конкретного объекта программы к физическому обогревателю необходима процедура (метод) инициализации. С учетом сказанного можно описать на языке Smalltalk интерфейс объекта-обогревателя следующим образом:

Heater methodsFor: 'initialize-release'

initialize: theLocation
realise

Heater methodsFor: 'modifiers'

turnOff
turnOn

Heater methodsFor: 'selectors'

isOn

Такое описание интерфейса в совокупности с пояснительной документацией на каждый из операторов интерфейса представляет все необходимое, что требуется для взаимодействия с любыми другими объектами программы.

Иначе обстоит дело с внутренним содержанием объекта-обогревателя. Вполне понятно, что для подключения обогревателя в сеть требуется электромеханическое реле, которое в свою очередь управляется сигналом, поступающим от компьютера через параллельный порт. Для включения обогревателя используется послылка байта, все разряды которого установлены в «1», а для отключения в «0». Исходя из этого, получим следующее описание реализации объекта-обогревателя:

Object subclass: #Heater

instanceVariableNames: 'thePort isOn'

```

classVariableNames: ' '
poolDictionaries: ' '
category: 'Hydroponics Gardening System'
Initialize: theLocation
    «Initialize the heater device driver by opening an RS232 port associated with the given location,
    theLocation is expected to be of the class Location.»

    thePort <— RS232Port open: theLocation.
    isOn <— false

release
    «Release the RS232 port associated with this heater.»

    thePort release.
    thePort <— nil

isOn
    «Return true if heater is on, false otherwise.»
    ^isOn

turnOff
    «Turn off the heater by writing a character with all bits reset to the RS232 port.»

    | aString |
    aString <— String new: 1.
    aString at: 1 put: (Character value: 0).
    thePort sendBuffer: aString.
    isOn <— false.
    aString release

turnOn
    «Turn on the heater by writing a character with all bits reset to the RS232 port.»

    | aString |
    aString <— String new: 1.
    aString at: 1 put: (Character value: 255).
    thePort sendBuffer: aString.
    isOn <— false.
    aString release

```

В соответствии с правилами Smalltalk две переменные (thePort и isOn) размещены с учетом ограничения к ним доступа. Если программист сделает попытку обратиться к этим переменным вне указанного класса, возникнет сообщение об ошибке.

Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательного порта управление должно осуществляться через область памяти. В такой ситуации нет необходимости изменять интерфейсную часть объекта, а достаточно переписать реализацию. Поскольку правила Smalltalk являются достаточно устаревшими, придется заново осуществить компиляцию измененного класса и других объектов, так как их поведение не изменяется, если только эти другие объекты не зависят от временных и пространственных характеристик прежнего кода (что крайне нежелательно и совершенно не нужно). При правильном использовании ограничение доступа позволяет локализовать те особенности

проекта, которые подвержены изменениям. По мере развития системы может оказаться, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют структуру объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать объемы используемой памяти. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов, связанных с данными.

В идеальном случае попытки обращения к данным, закрытым для доступа, должны выявляться во время компиляции программы. Вопрос реализации этих условий для конкретных языков программирования является предметом постоянных обсуждений. Мы уже видим, что Smalltalk обеспечивает защиту от прямого доступа к переменным другого класса, обнаруживая такие попытки во время компиляции. В тоже время Object Pascal такой возможности не предоставляет. Язык CLOS «занимает» в этом вопросе промежуточную позицию, возлагая все обязанности по ограничению доступа на программиста. В этом языке все слоты могут сопровождаться атрибутами, разрешающими чтение, запись и доступ к данным (в последнем случае чтение и запись). При отсутствии всех атрибутов слот является полностью защищенным от доступа. В языке C++ управление доступом и видимостью достигается с большей гибкостью. Элементы объекта могут быть отнесены к общедоступной, обособленной или защищенной части. Общедоступная часть «видима» для всех объектов; обособленная часть полностью закрыта для других объектов; защищенная часть «видима» только для данного класса и его подклассов.

Кроме того, в C++ существует понятие связанных классов (Friend), для которых обособленная часть является взаимодоступной. Защита информации является относительной: то, что защищено на одном уровне абстракции, является видимым на другом уровне. Только при явном неиспользовании разработчиком указаний возможностей и при умении воспользоваться возникающими трудностями можно обойти средства ограничения доступа. Ограничение доступа не гарантирует от глупости, как отметил Страуструп: «Защита гарантирует от вмешательства, но не от обмана» [53]. Разумеется, в этой книге нет таких примеров, когда невозможно узнать, как реализуется поведение объекта, но операционная система может ограничить доступ к конкретным файлам, которые описывают такое поведение. Практически не всегда можно познакомиться с тем, как реализован тот или другой класс, понятие его устройства, особенно при отсутствии эксплуатационной документации.

Модульность

Понятие модульности. По мнению Майерса, «Разделение программы на фрагменты позволяет частично уменьшить ее сложность... Однако гораздо важнее тот факт, что разделение программы улучшает проработку ее частей. Эти части весьма ценны для исчерпывающего понимания программы в целом» [54]. В некоторых языках программирования, например в Smalltalk, модульность не реализована и классы составляют лишь физическую основу декомпозиции. В других языках, включая Object Pascal, C++, Ada, CLOS, модульность является элементом конструкции и позволяет осуществлять на ее основе проектные решения. В таких языках классы и объекты составляют логическую структуру системы; эти абстракции организуются в модули, образуя физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

По Лискову, «модульность — это разделение программы на отдельно компилируемые фрагменты, имеющие между собой средства сообщения». Можно использовать также и определение Парнаса: «Связи между модулями — это предположения о возможности их использования» [55]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, также реализуется разделение интерфейсной части и реализации. Таким образом, модульность и ограничение доступа идут неотделимо друг от друга. В некоторых языках программирования модульность реализуется особыми приемами. Например, в языке C++ модулями являются отдельно компилируемые файлы. Для языков C/C++ традиционным является помещение интерфейсной части модулей в отдельные файлы с расширением .h (так называемые заголовочные файлы).

Реализация модуля описывается в файлах с расширением .c. Взаимосвязь файлов реализуется путем включения в них макроопределения #include. Такой подход строится исключительно на соглашениях и не является строгим требованием самого языка. В языке Object Pascal принцип модульности формализован несколько глубже. В этом языке определен особый синтаксис для интерфейсной части и реализация модуля, носящего имя Unit. В языке Ada модуль (называемый Package) также имеет две части — спецификацию и тело. Но в отличие от Object Pascal допускается раздельное определение связей модулей как для спецификаций, так и для тел. При этом тело модуля может быть связано с другими модулями, которые невидимы для спецификации.



Модульность позволяет размещать абстракции в различных разделах программы.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зелковиц, утверждая: «поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной» [56].

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Это такая же ситуация, которая возникает у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных вентилей типа ие, и-ие, или-ие, но можно и объединить группы таких вентилей в стандартные интегральные схемы, например, серий 7400, 7402 или 7404. Отсутствие стандартизованных фрагментов дает программисту гораздо большую степень свободы — как если бы электронщик имел в своем распоряжении средства создания кремниевых кристаллов.

Для небольших задач допустимо описание каждого класса и объекта в отдельном модуле. В других, наиболее тривиальных, случаях в один модуль лучше собрать все логически связанные классы и объекты и оставить незащищенными все те элементы, которые должны быть видны для всей программы. Однако такой подход применяется в исключительных случаях. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации механизма передачи сообщений. В больших системах, подобных описываемым в гл. 12, вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определение каждого класса сообщений в отдельном модуле. При этом не только возникает кошмарная ситуация с документированием, но чрезвычайно трудно для пользователя даже просто поиск нужных фрагментов описания. При введении в проект изменений потребуются модифицировать и перекомпилировать сотни модулей. Отсюда можно понять, что защита информации может обернуться и обратной стороной [57]. Деление программы на модули бессистемным образом гораздо хуже, чем отсутствие модульности вообще.

В традиционном структурном проектировании модульность связывается в первую очередь с логической группировкой подпрограмм на основе принципов взаимной связи и общей логики. В объектно-ориентированном программировании ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта и явно отличающиеся от подпрограмм.

На основе имеющегося опыта можно назвать приемы и правила, которые позволяют реализовать модули из классов и объектов наиболее эффективным образом. Бриттон и Парнас считают, что «конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна

быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для обеспечения процесса внесения изменений там, где они наиболее вероятны» [58]. На практике перекомпиляция тела модуля не является трудоемкой операцией: заново компилируется только данный модуль и компонуется вся программа. Перекомпиляция интерфейсной части модуля, наоборот, более трудоемка. В строго типированных языках приходится перекомпилировать интерфейс и тело самого измененного модуля, затем все модули, связанные с данным, модули, связанные с ними, и так далее по цепочке. В итоге для очень больших программ могут потребоваться многие часы на перекомпиляцию (если только оборудование не поддерживает фрагментарную компиляцию), что явно нежелательно. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была возможно более узкой (в пределах обеспечения необходимых связей). Стиль программирования требует сосредотачивать большую часть модуля в его теле. То, что при этом в интерфейсную часть выносится большой объем деклараций, не так опасно, как чрезмерный объем интерфейсного кода.

Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением ограничить доступ к данным и необходимостью обеспечения видимости тех или других абстракций в нескольких модулях. Парнас, Клементс и Вейс предложили следующее правило: «Особенности системы, подверженные изменениям следует, скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из другого модуля должен осуществляться только через процедуры данного модуля» [59]. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, приведем определение модульности:

Модульность — это свойство системы, связанное с возможностью декомпозиции ее на ряд тесно связанных модулей.

Таким образом, принципы абстрагирования, ограничения доступа и модульности являются взаимодополняющими. Объект определяет явные границы определенной абстракции, а ограничение доступа и модульность создают барьеры между абстракциями.

В процессе разделения системы на модули могут быть полезными два правила. Первое состоит в следующем: поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использоваться в системе многократно, распределение классов и объектов должно создаваться для этого максимальные удобства. Второе правило вытекает из того факта, что многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому размер модуля должен быть соответственно ограничен. Объявление функций в модуле может оказать большое влияние на возможность их вызова из другого модуля, так как это связано с разделением памяти на

страницы. Большое количество вызовов между сегментами загружает кэш-память и снижает характеристики всей системы.

Следует сказать о нескольких других моментах. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками проекта. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные — за остальную часть модулей. На более крупном уровне такие же соотношения справедливы для предприятий-соисполнителей. В последнем случае изменения в интерфейсе вызывают большое «напряжение» независимо от объема этих изменений, что оказывает сильное влияние на проектирование интерфейса. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу. К сожалению, иногда требования по документированию считаются главными при декомпозиции проекта (в большинстве случаев имея негативные последствия). Там, где один модуль требует большого объема документирования, делается десять модулей. Могут сказываться требования секретности: часть кода может быть несекретной, а другая — секретной; последняя в виде отдельного модуля (модулей).

Перечисленные факторы имеют множество взаимных связей, но главным является следующее: вычленение классов и объектов в проекте, а также организация модульной структуры — существенно независимые решения. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули — этап физического проектирования. Иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.

Примеры модульности. Посмотрим, как реализуется модульность в тепличном хозяйстве. Допустим, что для реализации интерфейса пользователя решено использовать стандартную рабочую станцию, а не какое-либо специальное оборудование. С помощью такой рабочей станции оператор может формировать новый план выращивания, модифицировать имеющиеся планы и наблюдать за исполнением плана. Для иллюстрации этой части системы можно воспользоваться языком Object Pascal как наиболее общедоступным. Ключевой абстракцией здесь является план выращивания. Необходимо, следовательно, создать модуль, связывающий все классы, относящиеся к планам выращивания. Основа такого модуля на Object Pascal будет выглядеть следующим образом:

```
unit UGrowingPlans; interface
...
implementation
  {$I UGrowingPlans.incl.p}
end.
```

Многообразие отмечена область деклараций, которая доступна для внешних модулей. Реализация классов, объектов и отдельных процедур этого модуля содержится в отдельном файле UGrowingPlans.incl.p.

Введем также модуль `UGardeningDialogs`, где будет собрано все, что относится к организации диалога с оператором. Поскольку этот модуль логически связан с классами модуля `UGrowingPlans`, его общая структура будет следующей:

```
unit UGardeningDialogs; interface
```

```
uses
```

```
    Memtype, QuickDraw, OSIntf, ToolIntf, PackIntf, CursorCtr,  
    UMAUnit, UViewCoords, UFailure, UMemory, UMenuSetup,  
    UObject, UList, UAssociation, UMacApp,  
    UGrowingPlans;
```

```
...
```

```
implementation
```

```
    ($I UGardeningDialogs.incl.p)
```

```
end.
```

Реализация данного модуля требует доступа к различным интерфейсам низкого уровня, что видно из описания интерфейсной части.

В проект может входить множество других модулей, таких, как `UGardeningCommands`, `UGardeningViews`, `UGardeningDocuments`, `UGardeningApplication`, каждый из которых использует интерфейс модулей более низкого уровня. В конечном счете мы должны определить основное тело всей программы, вызываемой из операционной системы. В объектно-ориентированном проектировании это является самым простым моментом, тогда как в структурном программировании тело программы составляет костяк и краеугольный камень, на который опирается все остальное. Объектно-ориентированный подход выглядит более естественно, как отметил Мейер: «Реальные программные системы более удобно описывать как совокупность сервисных механизмов. Как правило, можно описать систему с помощью отдельных функций, но слишком многое зависит от умения ... Для практической реализации систем нет предела» [60].

Иерархия

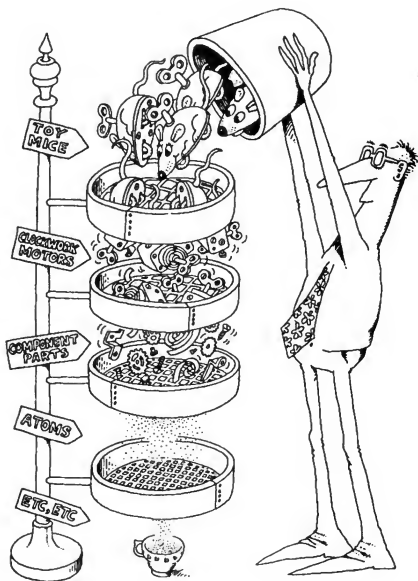
Понятие иерархии. Абстракция — вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает возможности их одновременного контроля. Ограничение доступа позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования иерархической структуры из абстракций. Определим иерархию следующим образом:

Иерархия — это ранжированная или упорядоченная система абстракций.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу).

Примеры иерархий: простое наследование. Важным элементом объектно-ориентированных систем и основным видом иерархии по номенклатуре является упоминавшаяся выше концепция наследования. Наследование означает такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких других



Абстракции организуются иерархически.

классов (соответственно простое и множественное наследование). Иными словами, наследование — такая иерархия абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов.

Рассмотрим, например, различные виды растений, выращиваемых в теплицах. Выше уже была рассмотрена обобщенная абстракция плана выращивания растений. Однако для оптимизации урожая план должен быть специализирован в зависимости от культур (например, для овощей, фруктов и цветов). Это означает, что стандартный план выращивания фруктов является разновидностью плана выращивания дополненного особенностями, например сроками созревания. Подобные соотношения могут быть определены на языке C++ следующим образом:

```
class StandardFruitGrowingPlan: public GrowingPlan {
public:
    StandardFruitGrowingPlan ();
    StandardFruitGrowingPlan (const StandardFruitGrowingPlan&);
    virtual ~StandardFruitGrowingPlan ();

    virtual int daysUntilHarvest (day currentDay) const;

private:
    day timeToHarvest;
};
```

Из приведенного определения видно, что класс StandardFruitGrowingPlan точно повторяет свой суперкласс Growing Plan за несколькими исключениями. В подкласс введен новый элемент структуры (timeToHarvest), изменены конструктор и деструктор, добавлена виртуальная функция (daysUntilHarvest). Используя этот новый класс уже в качестве суперкласса, можно определить еще более специализированные подклассы, например для плана выращивания яблок.

Для установленной иерархии наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии обобщения-специализации. Суперклассы при этом отражают наиболее общие, а подклассы — более специализированные абстракции, которые могут быть одновременно дополнены, модифицированы и даже защищены. Принцип наследования позволяет упростить выражения абстракций, делает проект менее громоздким и более выразительным. Кокс пишет: «В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться «с нуля». Классы лишаются взаимоотношений, и их методическая часть реализуется каждым программистом по-своему. Стройность системы может быть достигнута только на основе дисциплины программистов. Принцип наследования позволяет создавать новые программы и обучать новичков на основе уже готовых продуктов» [61].

Принципы абстрагирования, ограничения доступа и иерархии конкурируют между собой. Даифорт и Томлисон утверждают: «Абстрагирование данных состоит в установлении жестких границ, защищающих состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов» [62]. Для любого класса обычно существуют два вида объектов-пользователей: объекты, которые используют операции данного класса для доступа к его элементам, и объекты-подклассы, полученные наследованием данного класса. Лисков

считает, что существуют три способа нарушения механизма ограничения доступа через механизм наследования: «подкласс может получить доступ к данным своего суперкласса, осуществить вызов обособленной (защищенной) функции суперкласса и обратиться напрямую к суперклассу своего суперкласса» [63]. Различные языки программирования по-разному реализуют механизмы наследования и ограничения доступа, наиболее гибким в этом отношении является C++. В нем интерфейсная часть класса может быть разделена на три части: обособленную — видимую только для самого класса, защищенную — видимую также и для подклассов; общедоступную — видимую для всех.

Множественное наследование. В предыдущем примере рассматривалось простое наследование, когда подкласс создается только из одного суперкласса. В ряде случаев полезно реализовать наследование от нескольких суперклассов. Предположим, что нужно определить класс, представляющий разнообразия растений. Используем для этого язык CLOS и получим следующее:

```
(defclass plant ()
  ((name :initarg :name
         :reader plant-name)
   (date-planted :initarg :date-planted
                 :reader date-planted)
   (germination-time :initarg :germination-time
                     :reader germination-time)
   (actual-germination :initform nil
                       :assessor actual-germination))
  (:documentation «The base class of all plants.»))
```

Структура такого объекта состоит из четырех слотов (name, date-planted, germination-time, actual-germination). Для каждого слота определены методы инициализации (:initarg и :initform), для первых трех слотов — методы чтения данных (:reader), а для четвертого слота — методы чтения и записи (:assessor).

Мы уже определили, что в тепличном хозяйстве выращивание цветов, овощей и фруктов имеет свою специфику. Например, для цветов важно знать момент зацветания, а для фруктов момент сбора урожая. Чтобы реализовать указанные требования, необходимо из данного класса Plant образовать два новых класса — Flowering-plant и Fruit/vegetable-plant. А что если потребуется план, моделирующий сразу и выращивание цветов, и созревание фруктов? Цветоводы иногда используют цветы яблонь, вишни и слив. Тогда потребуется создать третий класс, включающий оба предыдущих подкласса. Но лучший путь решения указанной проблемы без подобной избыточности — множественное наследование. Для этого создаются отдельно классы для овощей, фруктов и цветов:

```
(defclass flowering-plant-mixin ()
  ((time-to-flower :initarg :time-to-flower
                  :reader time-to-flower)
   (time-to-seed :initarg :time-to-seed
                 :reader time-to-seed))
  (:documentation «A mixin class for flowering plants.»))

(defclass fruit/vegetable-plant-mixin ()
  ((time-to-harvest :initarg :time-to-harvest
                   :reader time-to-harvest))
  (:documentation «A mixin class for fruits and vegetables.»))
```

Эти классы самостоятельны и не имеют суперкласса. Они предназначены для последующего смешения между собой с целью создания подклассов. Например, подкласс для выращивания цветов будет следующим:

```
(defclass flowering-plant (plant flowering-plant-mixin)
  ()
  (:documentation «A flowering plant class.»))
```

а для фруктов и овощей другим:

```
(defclass fruit/vegetable-plant (plant fruit/vegetable-plant-mixin)
  ()
  (:documentation «A fruit or vegetable plant.»))
```

Во всех случаях подкласс образован наследованием от двух суперклассов. Структура полученных подклассов содержит все необходимые элементы. Теперь определим класс для фруктов и цветов одновременно:

```
(defclass flowering/fruit/vegetable-plant (plant
                                           flowering-plant-mixin
                                           fruit/vegetable-plant-mixin)
  ()
  (:documentation «A flowering fruit or vegetable plant .»))
```

Агрегатирование. Если иерархия по номенклатуре определяет отношение обобщения-спецификации, то иерархия по составу определяет отношения агрегатирования. Например, приведенный выше объект состоит из шести меньших объектов (четыре слота из класса `plant` и два из класса `flowering-plant-mixin`). Для таких иерархических структур часто употребляется термин «уровни абстракций», впервые введенный в работе [64]. Для иерархии по номенклатуре более высокому уровню соответствуют более общие абстракции, а более низкому — специализированные. В упомянутом примере класс `plant` является абстракцией более высокого уровня, чем `flowering-plant`. Для иерархии по составу более высокий уровень представляют те абстракции, которые используют в своем составе другие классы. Следовательно, класс `StandardFruitGrowingPlan` более высокого уровня, чем тип `day`, входящий в его состав.

Типизация

Понятие типизации. Концепция типизации строится на понятии типов абстрактных данных. По определению Дейтча: «Тип — это точное определение свойств строения или поведения, которое присуще некоторой совокупности объектов» [65]. Далее для удобства термины «тип» и «класс» будут использоваться в качестве эквивалентов¹⁾. Несмотря на схожесть понятий «класс» и «тип», в качестве отдельного элемента объектного подхода выделяется типизация, поскольку эта концепция подчеркивает различные особенности абстракций. Определим типизацию следующим образом:

1) Тип и класс — не одно и то же; в некоторых языках эти два понятия различаются. Например, ранние версии языка Trellis/Owl позволяют объекту иметь как класс, так и тип. Даже в языке Smalltalk объекты классов `SmallInteger`, `LargeNegativeInteger` и `LargePositiveInteger` принадлежат одному и тому же типу `Integer` но разным классам [66]. Для большинства людей тем не менее разделение понятий типа и класса крайне неудобно и приносит мало пользы. Достаточно сказать, что класс реализует тип.

Типизация — это ограничение, накладываемое на класс объектов и препятствующее взаимозамене различных классов (или сильно сужающее возможность такой взаимозамены).

Типизация позволяет выполнять описание абстракций таким образом, что реализуется поддержка проектных решений на уровне языка программирования. О важности такого вида поддержки для программирования на уровне больших проектов заметил Вегнер [67]. В то же время объектные и объектно-ориентированные языки программирования могут быть строго типизированы, нестрого и совсем не типизированы, что позволяет говорить о типизации как о второстепенном элементе.

Примеры строгой и нестрогой типизации. Возвращаясь к примеру реализации интерфейса пользователя в тепличном хозяйстве, предположим, что имеется следующее описание классов (неполное) на языке Object Pascal:

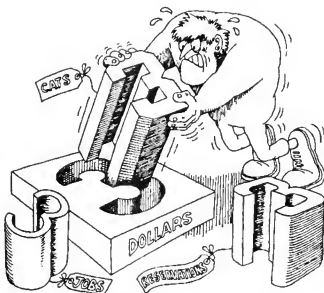
```
TShape = object (TObject)
    fPosition: Point;
    procedure TShape.Draw (Area: Rect);
    function TShape.IsVisible: Boolean;
end;

TText = object (TShape)
    fValue: Str255;
    procedure TText.Draw (Area: rect); override;
end;

TGreenhouse = object (TShape)
    fHydroponics_Tanks: array[1..10] of ThydroponicsTank;
    procedure TGreenhouse.Initialize;
    procedure TGreenhouse.Draw (Area: rect); override;
end;
```

В языке Object Pascal все объявленные в интерфейсной части класса поля и операции являются общедоступными. Следовательно, реализация операций такого класса является «видимой». Это дает основания говорить о неполной реализации в Object Pascal концепции ограничения доступа.

В качестве суперкласса для ряда объектов, связанных с обслуживанием экрана рабочей станции, введен класс TShape путем наследования базового класса TObject. Более специализированными подклассами, объекты которых являются изображениями на экране, служат TText и TGreenhouse. В классе TShape введен общий метод Draw, так как все объекты связаны с изображениями; в подклассах TText и TGreenhouse этот метод доопределен для вывода текстов и изображения теплицы соответственно.



Строгая типизация ограничивает перечень абстракций, которые могут быть использованы в конкретной процедуре.

Object Pascal является строго типизированным языком и требует явного определения типа каждой переменной, параметра и поля в описании класса. Предположим следующий набор определений:

```
AnObject      : TObject;
AShape        : TShape;
ATextString   : TText;
AGreenhouse   : TGreenhouse;
```

Теперь можно указать операторы для образования новых объектов:

```
new (AnObject);
new (AShape);
new (ATextString);
new (AGreenHouse);
```

Переменные, подобные ATextString, не являются объектами. ATextString — это только имя для обозначения объекта класса TText: когда говорят «Объект ATextString», в действительности имеют в виду экземпляр объекта TText, отмеченный переменной ATextString. Эта особенность будет подробно рассмотрена в следующей главе.

Строгая типизация в Object Pascal требует проверки во время компиляции на соответствие типов в операторах вызова процедур. Ниже приведен пример правильного написания операторов:

```
AShape.Draw (SomeArea);      {Draw is defined for the class TShape}
ATextString.Draw (SomeArea); {Draw is defined for the class TText}
```


Следующие операторы содержат ошибки и не смогут пройти через процесс компиляции:

```
AnObject.Draw (SomeArea); {illegal}
ATextString.Initialize;    {illegal}
```

Ошибки в этих операторах заключаются в том, что методы Draw и Initialize не определены для соответствующих переменных ни в их классах, ни в суперклассах. А следующий оператор написан верно:

```
if AGreenhouse.IsVisible then
    ...
```

Хотя метод IsVisible не определен для класса TGreenhouse, но он определен в его суперклассе TShape. Рассмотрим тот же пример на инициализированном языке Smalltalk.

Переменные в следующем определении являются инициализированными:

```
! anObject aShape ATextString aGreenhouse !
```

Следующий оператор

```
anObject draw: SomeArea.
```

пройдет через компиляцию, но точный его смысл останется неясным до запуска программы на исполнение. Если переменная anObject окажется связанной с классом Shape (имеющий в своем описании метод draw), то программа будет выполнена без ошибок. И наоборот, если anObject окажется связанной с экземпляром класса Bad (предопределенного в Smalltalk и не имеющего метода draw в своем описании), то возникнет ошибка при выполнении программы.

Строго типизированными являются такие языки, в которых все выражения проходят проверку на соответствие типов. В описании ниже примере, иллюстрируется смысл соответствия типов на основе сделанных ранее определений на Object Pascal. Следующие два оператора верны:

```
AnObject := AnObject;
AShape := ATextString;
```

В первом операторе класс переменной левой части соответствует классу выражения правой части. Во втором операторе класс переменной левой части (TShape) является суперклассом переменной правой части (TText). Рассмотрим другие два оператора:

```
AGreenhouse := AShape; {illegal}
ATextString := AGreenhouse; {illegal}
```

Оба оператора ошибочны, так как класс переменной в левой части операторов является подклассом переменной в правой части. Иногда возникает необходимость преобразовать переменную одного типа в переменную другого типа. Например, для предопределенного в библиотеке Object Pascal класса TList имеется операция Each, позволяющая получить доступ к любому элементу списка:

```
procedure TList.Each (procedure DoToItem (Item: TObject));
```

5 Гради Буч

Если известно заранее, что список всегда состоит из объектов класса TGreenhouse, то можно реализовать явную связь значений двух типов, как в следующем ниже операторе присвоения:

```
procedure DoToItem (Item : TObject);
begin
    ...
    AGreenhouse := TGreenhouse (Item);
    ...
end;
```

Этот оператор удовлетворяет требованиям соответствия типов, но не гарантирует абсолютного соответствия. Если, например, во время исполнения программы в списке окажется объект класса TText, то возникнет ошибка.

Теслер [68] отметил следующие важные преимущества строго типизированных языков:

- * «Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.
- * В большинстве случаев процесс повторного редактирования-компиляции-отладки достаточно утомителен и раннее обнаружение ошибок является обязательным условием.
- * Декларирование типов упрощает документирование программ.
- * Многие компиляторы позволяют генерировать более эффективный код при явном определении типов».

Язык, в которых типизация отсутствует, обладают большей гибкостью, но даже для таких языков, по мнению Борнинга и Ингалса: «в большинстве случаев программисты фактически знают, какие объекты выступают в качестве аргументов и какие будут возвращаться» [69]. На практике, для больших систем, надежность языков со строгой типизацией компенсирует некоторую потерю в гибкости по сравнению с истипизированными языками.

Статическая и динамическая связь. Концепции строгой и статической типизации по существу различны. Строгая типизация имеет отношение к контролю соответствия типов, а статическая (иначе называется статической или ранней связью) — имеет отношение ко времени, когда имена связываются с типами. Статическая связь означает неизменность типов всех переменных и выражений во время компиляции; динамическая связь (называемая также поздней связью) означает ситуацию, когда тип всех переменных и выражений определяется только во время исполнения программы. Концепции типизации и связей являются независимыми, поэтому язык программирования может быть строго типизирован со статической связью (Ada) и с динамической связью (Smalltalk). Язык CLOS занимает промежуточное положение между C++ и Smalltalk, так что при реализации определения, сделанные программистом, могут быть либо приняты, либо не приняты. Поясним сказанное на примере, в котором используется Object Pascal. Выше уже использовался класс TList из библиотеки Object Pascal, представляющий собой связанный список. Определим этот класс следующим образом:

```
TList = object (TObject)
    ...
    procedure TList.InsertFirst (Item: TObject);
    procedure TList.Each (procedure DoToItem (Item: TObject));
end;
```

Здесь использованы два метода: один для включения в список нового элемента (модификатор), другой для доступа к отдельным элементам списка (итератор). Так как Object Pascal поддерживает динамические связи, список может быть либо однородным (все элементы одного определенного класса), либо разнородным (из элементов разных классов) при условии, что каждый элемент списка является реализацией класса TObject или любого его подкласса.

Предположим, что имеется объект класса TList с именем AList, представляющий собой разнородный список объектов класса TShape и производных от него подклассов. Тогда можно записать следующие операторы:

```
AList.InsertFirst (AShape);
AList.InsertFirst (ATextString);
AList.InsertFirst (AGreenhouse);
```

Эти операторы отвечают принципу соответствия типов, так как действительные параметры в каждом из них являются объектами того же класса (или подклассом) соответствующего формального параметра (TObject).

Предположим, что нужно написать процедуру вывода на экран произвольного объекта из такого списка. Например,

```
procedure Draw_Item (Item: TObject);
begin
    TShape (Item).Draw (SomeArea);
end;
...
AList.Each (Draw_Item);
```

Обращение к операции Each активизирует итератор списка, который в свою очередь обращается к процедуре Draw_Item. Отметим, что в процедуре Draw_Item переменная Item связана с классом TShape и его подклассами, что позволяет обратиться к процедуре Draw. Однако во время компиляции не известно, какой подкласс связан с объектом, обозначенным формальным параметром Item: это могут быть, в частности, классы TText или TGreenhouse. Это пример динамической связи.

Гарантию того, что формальный параметр Item будет связан с классом TShape, создает контроль занесения объектов в список. Поскольку для класса TShape определен метод Draw, вызов этой процедуры отвечает условиям согласования типов. Таким образом, вызов итератора Each приводит к прохождению по списку и к вызову процедуры Draw для каждого элемента списка. Поскольку объекты списка могут быть разными (относиться к разным классам), то и вызов метода Draw должен выполняться по-разному. В конечном счете только при выполнении программы обнаружится, какая из процедур Draw будет действительно выполнена. Это свойство называется полиморфизмом: определенное имя (объявленная переменная) может означать объект любого класса, относящегося к определенному суперклассу. Любой из объектов, связанных с таким именем, должен выполнять некоторое множест-

во общих операций [70]. Противоположностью полиморфизма является мономорфизм, свойственный всем языкам со строгой типизацией и статическими связями, например языку Ada.

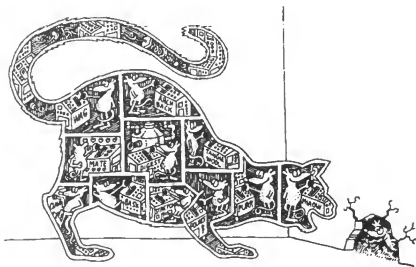
Полиморфизм возникает на стыке принципов наследования и динамических связей. Это свойство является самым существенным в объектно-ориентированном программировании наряду со свойством реализации абстракций. Именно это свойство отличает ООР от более традиционных методов программирования с использованием типов абстрактных данных. Кроме того, как мы увидим ниже, полиморфизм является важной концепцией в объектно-ориентированном проектировании.

Параллелизм

Понятие параллелизма. Для определенной категории задач автоматические системы реализуют обработку многих событий, происходящих одновременно. В других случаях время, затрачиваемое на вычисления, превышает ресурсы одного процессора. В обеих ситуациях естественно предложить использование нескольких компьютеров для решения общей задачи или для реализации многозадачного режима обработки. Единичный процесс (известный также как канал управления) — это путь, начало которого находится в том месте системы, где возникает некоторое независимое динамическое воздействие. Любая программа включает по меньшей мере один канал управления, но в параллельной системе таких каналов может быть несколько: один могут быть временными, а другие могут сохраняться в течении всего времени выполнения программы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

Лим и Джонсон утверждают: «Реализация параллельности в объектно-ориентированных языках такая же, как в любых других языках, — параллелизм не связан с ООР на самом нижнем уровне абстракции» [71]. Что касается создания больших программных систем, охватывающих множество каналов управления, то здесь ситуация гораздо сложнее: можно упустить из внимания тупики, блокировки, исчерпание процесса, взаимные противоречия и временные ограничения. К счастью, как отмечают те же авторы: «на верхних уровнях абстракции ООР позволяет упростить решение вопросов параллелизма для большинства программистов, ограничивая параллелизм внутри множества абстракций» [72]. Блэк и другие авторы сделали следующий вывод: «объектный подход удовлетворяет требованиям распределенных систем, поскольку неявно определяет блоки распределения и перемещения и взаимодействующие объекты» [73].

В то время как объектно-ориентированное программирование строится на абстракции данных, ограничении доступа и наследовании, параллелизм связан с абстрагированием процессов и синхронизацией [74]. Объект является основой, которая объединяет обе концепции: каждый объект (как абстракция реальности) может представлять собой отдельный канал управления (абстракцию процесса). Такой объект называется *активным*. Для систем, построенных на основе OOD, реальность может быть представлена как совокупность взаимодействующих объектов, часть которых является активной и выступает в роли узлов независимых действий. На этой основе дадим следующее определение параллелизма:



Параллелизм характеризует возможность одновременного функционирования объектов.

Параллелизм — свойство объектов находиться в активном, либо пассивном состоянии.

Примеры параллелизма. Рассмотрим абстракции, определенные на языке Ada, для последовательности классов, представляющих датчики температуры. Мы уже говорили о классе активных датчиков, которые периодически измеряют температуру и посылают сообщение другому объекту при изменении температуры на определенную величину. Чтобы показать с помощью языка Ada механизм описания параллельных процессов, следует дополнить определение класса `Air_Temperature_Sensor` следующим образом (не рассматривая здесь операцию калибровки):

```
task type Air_Temperature_Sensor is
  entry Initialize      (Its_Location      : in Location;
                        Lower_Alarm_Limit : in Temperature;
                        Upper_Alarm_Limit : in Temperature);
end Air_Temperature_Sensor;
```

Для каждого элемента в такого рода задачах создается новый процесс. С точки зрения объектной ориентации над подобными объектами можно выполнять только одно действие. В частности, такой объект можно инициализировать, сообщив ему место расположения, верхнюю и нижнюю температурные границы (вне которых действуют другие объекты через формальный параметр `Temperature_Alarm`). Допустим, что обмен информацией датчики осуществляют через определенную область памяти.

На языке Ada указанные выше решения будут выражены в теле пакета **Air Temperature Sensors** следующим образом:

```
type Word is range (2**15 -1) .. (2**15 -1);
for Word'Size use 16;
```

```
Sensor_Memory_Map: array(Location) of Word;
for Sensor_Memory_Map use at 16#377FF0#;
```

```
Time_Interval: constant Duration:=2.0;
```

Введенные декларации являются защищенными как скрытая часть абстракции и потому, что отведенная для обмена данными область памяти начинается с 16-ричного адреса 377FF0, а под каждый датчик отведено 16-разрядное число. Константа Time_Interval определяет периодичность чтения данных из физического датчика (в данном случае один раз в 2 с). Суть этой задачи состоит в чтении каждые 2 с данных из определенной области памяти и сообщении другим объектам о наличии изменений или об аварийной ситуации. Без учета вопросов калибровки и конечных условий тело задачи будет выглядеть следующим образом:

```
task body Air_Temperature_Sensor is
  Current_Location      : Location;
  Sensor_Value          : Word;
  Current_Temperature    : Temperature;
  Previous_Temperature   : Temperature:=Temperature'Last;
  Lower_Limit           : Temperature;
  Upper_Limit           : Temperature;
  Next_Time             : Calendar.Time:=Calendar.Clock;
begin
  accept Initialize :
    (Its_Location      : in Location;
     Lower_Alarm_Limit : in Temperature;
     Upper_Alarm_Limit : in Temperature) do
    Current_Location      : Its_Location;
    Lower_Limit          : =Lower_Alarm_Limit;
    Upper_Limit          : =Lower_Alarm_Limit;
  end Initialize;
  loop
    delay (Next_Time - Calendar.Clock);
    Sensor_Value:=Sensor_Memory_Map (Current_Location);
    Current_Temperature:= Temperature (Float (Sensor_Value) *Temperature'Delta);
    if (Current_Temperature /=Previous_Temperature) then
      if (Current_Temperature<Lower_Limit) or
         (Current_Temperature>Upper_Limit) then
        Temperature_Alarm (Current_Location, Current_Temperature);
      else
        Previous_Temperature:=Current_Temperature;
        Temperature_Has_Changed (Current_Location, Current_Temperature);
      end if;
    end if;
    Next_Time:=Next_Time + Time_Interval;
  end loop;
end;
```

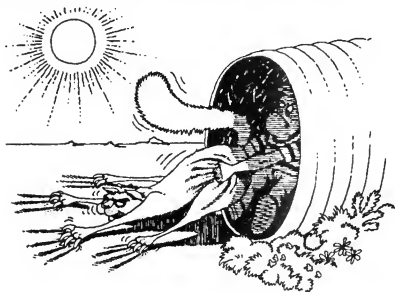
Для большей ясности этот пример содержит некоторое избыточное число локальных переменных, чем необходимо в действительности. После инициализации активного объекта его процесс повторяется периодически через каждые несколько секунд в течение всего времени чтения данных из физи-

ческого датчика. Если новое прочитанное значение отличается от предыдущего, алгоритм не прерывается. Но если новое значение вышло за установленные пределы, вызывается процедура `Temperature_Alarm`. В остальных случаях для сообщения об изменении температуры другими объектами используется процедура `Temperature_Has_Changed`.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения между активными объектами, а также с остальными объектами, действующими последовательно. Например, следует предусмотреть разрешение ситуаций, когда два объекта одновременно посылают сообщение третьему, чтобы предотвратить смешение данных. При наличии параллелизма недостаточно просто описать методическую часть, необходимо предусмотреть в ней способы защиты для случая многоканального управления.

Существуют экспериментальные параллельные объектно-ориентированные языки программирования (`Actor`, `Orient 84/K`, `ABCL/1`), имеющие механизмы активизации и синхронизации. Более подробные сведения о них и о ряде других приведены в приложении. Из числа языков, использованных в данной книге, многозадачный режим реализуют прямо только `Smalltalk` и `Ada` (`Smalltalk` содержит класс `Process` для этой цели, а `Ada` реализует тип «task»).

В `C++` параллельные объекты могут быть реализованы в системе `Unix` вызовом `fork`. `Object Pascal` и `CLOS` обычно используются только для последовательных задач и не имеют механизмов распараллеливания.



Устойчивость позволяет объектам сохранять свое состояние и принадлежность к определенному классу.

Устойчивость

Любой объект в программе занимает определенное место и существует в течение определенного времени. Аткинсон и другие выдвинули гипотезу о континууме способов существования объектов, начиная с преходящих объектов, которые существуют лишь во время определенных вычислений, и кончая объектами в базах данных, которые существуют даже вне пределов программы. В этот спектр, определяющий устойчивость объектов, можно включить следующие виды:

- * «Промежуточные результаты вычисления выражений.
- * Локальные переменные в вызове процедур.
- * Собственные переменные (например, в ALGOL-60), глобальные переменные и главные элементы.
- * Данные, сохраняющиеся между вызовами основной программы.
- * Данные, остающиеся без изменений в разных версиях программы.
- * Данные, которые переживают всю программу» [75].

Традиционные языки программирования, как правило, реализуют только три первых уровня устойчивости, а последние три обычно связываются с технологией баз данных. Это приводит к неожиданным решениям: программисты разрабатывают специальные схемы для сохранения объектов в период между запуском программы, а конструкторы баз данных адаптируют свою технологию под временно живущие объекты [76].

Унификация принципов параллелизма для объектов позволила бы в этом случае подняться до уровня требований объектно-ориентированного программирования. Аналогичным образом реализация принципа устойчивости в объектном подходе поднимает его на уровень объектно-ориентированных баз данных (OODB). На практике подобные базы данных строятся на хорошо проверенных принципах, таких, как последовательные, индексированные, иерархические, сетевые или реляционные модели. Для программиста они реализуют абстракцию объектно-ориентированного интерфейса, благодаря чему упрощается доступ к базе данных и другие операции с объектами, время существования которых превышает время функционирования отдельной программы, унифицируются. Эта унификация значительно упрощает разработку отдельных видов программ, позволяя, в частности, применить единый подход к разным сегментам программы, одни из которых связаны с базами данных, а другие не имеют такой связи.

Число OODB очень ограничено, это TAXIS, SDM, DAPLEX и GEM [77]. Ни один из пяти языков программирования, упоминаемых в данной книге, не поддерживает прямо концепцию устойчивости, поэтому не приводятся и соответствующие примеры. Однако в гл. 9 и 10 будет показано, что имеется возможность имитации устойчивости на известных языках программирования.

Устойчивость — понятие, связанное не только с временем существования данных. В OODB сохраняется не только состояние объекта, но и способ интерпретации класса любой другой программы должен быть определен однозначно. Из этого можно понять всю сложность управления базой данных в процессе ее наращивания, в частности при изменении классов объектов.

Далее рассматривается только устойчивость во времени. В большинстве систем в течение всего времени существования объекты занимают определенное место в физической памяти. Для многопроцессорных систем можно рассматривать и вопрос устойчивости в пространстве. В таких системах необходимо рассмотреть перемещение объекта между процессорами и даже изменение при этом способа представления объектов. Эта разновидность устойчивости рассмотрена в гл. 12. В заключение дадим следующее определение устойчивости:

Устойчивость — свойство объекта существовать во времени (вне зависимости от процесса, породившего данный объект) и (или) в пространстве (перемещение объекта из адресного пространства, в котором он был создан).

2.3. ПРИМЕНЕНИЯ ОБЪЕКТНОГО ПОДХОДА

Преимущества объектного подхода

Как уже говорилось выше, объектный подход принципиально отличается от тех подходов, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектный подход требует отказа от всех ранее найденных и испытанных методов и приемов, напротив, новые элементы всегда основываются на предшествующем опыте. Объектный подход создает множество существенных удобств, которые при других условиях не могут быть обеспечены. Наиболее важным является то, что объектный подход позволяет создавать системы, которые воплощают пять атрибутов хорошо структурированных сложных систем. Кроме того, можно назвать еще пять преимуществ, которые связаны с использованием объектного подхода.

Во-первых, объектный подход позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Строустрап отмечал: «Не всегда очевидно, как в полной мере использовать преимущества такого языка, как C++. Существенно повысить эффективность и качество кода можно просто за счет использования C++ в качестве «улучшенного C» с элементами абстракции данных. Однако гораздо более значительным достижением является использование иерархий классов в процессе проектирования. Именно это называется OOD и именно здесь преимущества C++ продемонстрированы в наибольшей степени» [78]. Опыт показал, что использование таких языков, как Smalltalk, Object Pascal, C++, CLOS и Ada, без элементов объектного подхода является малоэффективным.

Во-вторых, использование объектного подхода существенно повышает качество разработки в целом и ее фрагментов [79]. Объектно-ориентированные системы часто получаются более компактными, чем не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта и большее удобство в планировании разработок.

В-третьих, использование объектного подхода приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке в случае существенных изменений исходных требований.

В гл. 7 показано, что объектный подход уменьшает риск в разработке сложных систем прежде всего потому, что процесс интеграции растягивается во времени жизненного цикла системы. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектный подход ориентирован на человеческое восприятие мира, или, по словам Рабсона, «многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам» [80].

Авиационное оборудование
Автоматизация учреждений
Автоматизированное проектирование
Автоматизированное обучение
Автоматизированное производство
Базы данных
Гипериосители
Компоненты программного обеспечения
Контроль программного обеспечения
Математический анализ
Медицинская электроника
Моделирование авиационной и космической техники
Музыкальная композиция
Написание сценариев
Нефтяная промышленность
Обработка коммерческой информации

Оживление
Операционные системы
Планирование инвестиций
Подготовка документов
Программные средства космических станций
Проектирование интерфейса пользователя
Проектирование СБИС
Распознавание образов
Робототехника
Системы телеметрии
Системы управления и регулирования
Средства разработки программ
Телекоммуникации
Управление воздушным движением
Управление химическими процессами
Экспертные системы

Примеры использования объектного подхода

Возможность использования объектного подхода доказана для задач самого разного характера. Ниже приведен перечень областей применения, для которых реализованы объектно-ориентированные системы. Более подробные сведения для этих и других случаев можно найти в литературе, приведенной в библиографии.

В настоящее время OOD — единственная методология, позволяющая справиться со сложностью, присущей самым большим системам. Однако, следует заметить, что в ряде случаев применение OOD может оказаться нецелесообразным по принципиальным мотивам, как, например, неподготовленность персонала или отсутствие соответствующих технических средств.

Вопросы, требующие дальнейшего исследования

Для более эффективного использования объектного подхода, предстоит ответить на следующие вопросы:

- Что же такое классы и объекты?

- * Как правильно идентифицировать классы и объекты для конкретного приложения?
 - * Оптимальная система формального выражения решений, принимаемых в процессе OOD?
 - * Что может помочь нам в процессе структурирования объектно-ориентированных систем?
 - * Как организовать управление процессом OOD?
- Этим вопросам посвящены следующие пять глав.

Заключение

- * Развитие программной технологии привело к созданию методов объектно-ориентированного анализа, проектирования и программирования, которые направлены на решение задачи программирования больших систем.
- * Существует ряд приемов программирования: процедурно-ориентированное, объектно-ориентированное, логически-ориентированное, ориентированное на правила и ориентированное на ограничения.
- * Объектный подход образует концептуальную основу для объектно-ориентированной методологии; он включает принципы абстрагирования, ограничения доступа, модульности, иерархии, типизации, параллелизма и устойчивости.
- * Абстрагирование означает выделение таких существенных характеристик объекта, которые отличают его от всех других объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего его рассмотрения.
- * Ограничение доступа — процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта как целого.
- * Модульность — свойство системы, связанное с возможностью ее декомпозиции на ряд тесно связанных модулей.
- * Иерархия — ранжированная или упорядоченная система абстракций.
- * Типизация — ограничение, предъявляемое классу объектов, препятствующее взаимозамене различных классов и в большинстве случаев сильно сужающее возможность такой взаимозамены.
- * Параллелизм — свойство, отличающее активные объекты от неактивных.
- * Устойчивость — свойство объектов существовать во времени и (или) в пространстве.
- * Применение объектного подхода позволяет создавать системы, обладающие пятью атрибутами хорошо структурированных сложных систем.

Дополнительная литература

Принципы объектного подхода впервые установлены в следующих работах: Jones [F, 1979], Williams [F, 1986]. В работе Kay Ph.D [F, 1969] определены направления развития OOP. Анализ механизма абстрагирования для программирования высокого уровня выполнен Shaw [J, 1984]. Теоретические основы абстрагирования можно найти в работах Liskow, Guttag [H, 1986], Guttag [J, 1980] и Hilfinger [J, 1982]. Parnas [F, 1979] заложил основы защиты информации. Особенности и роль иерархии приведены в работе Pattee [J, 1973].

Имеется обширная литература по OOP. Хороший обзор объектных и объектно-ориентированных языков сделан Cardelli, Wegner [J, 1985] и Wegner [J, 1987]. Основы OOP хорошо освещены в следующих работах:

Stejic, Bobrow [G, 1986], Stroustrup [G, 1988], Nygaard [G, 1986]. Их развитие можно найти у Cox [G, 1986], Meyer [F, 1988], Schmucker [G, 1986], Kim, Lochovsky [F, 1989]. Booch [F, 1981, 1982, 1986, 1987, 1989] формализовал методы OOD. Варианты этих методов можно найти в HOOD [F 1987] применительно к проекту Европейской космической станции, а также у Seidewitz, Stark, [F, 1988] GOOD.

Подробная методика предлагалась Wirfs-Brock, Wilkerson [F, 1989] (отмечается значение реагирования), Constantine [F, 1989] и Wasserman [F, 1989]. Сюда же следует отнести работу Ross [F 1987] по моделированию в целом, работы Abelson, Sussman [H 1985] по общим вопросам программирования. Методы объектно-ориентированного анализа изложены Shlaer, Mellor [B, 1988], Bailin [B, 1988] и Coad, Yourdon [B, 1990].

Хорошие статьи по всем вопросам объектно-ориентированного подхода можно найти в работах Peterson [G, 1987], Schriver, Wegner [G, 1987]. Обширный материал по этой теме представляют конференции последних лет по объектно-ориентированным методам в компьютерной технике (ООС). Наиболее интересны конференции USENIX C++, OOPSLA (объектно-ориентированные системы программирования, языки, практические приложения), ECOOP (Европейская конференция по ООР).

Глава 3

Классы и объекты

Использование объектно-ориентированной методологии для создания сложных программных систем подразумевает наличие базовых строительных блоков в виде классов и объектов. Выше было дано неформальное определение этих двух элементов. В данной главе природа классов и объектов рассматривается более подробно.

3.1. ОБЪЕКТ

Что является объектом

Способностью к распознаванию объектов физического мира человек обладает с раннего возраста. Ярко окрашенный мяч привлекает внимание ребенка, но, если спрятать мяч, ребенок, как правило, не пытается его искать: как только предмет покидает поле зрения, он перестает для него существовать. Только в возрасте около одного года у ребенка появляется представление о предмете. Покажите мяч годовалому ребенку и спрячьте его: обычно ребенок начинает искать спрятанный предмет. Ребенок связывает понятие предмета с постоянством и индивидуальностью независимо от действий над этим предметом [1].

Выше объект был определен как осязаемая реальность, имеющая четко определяемое поведение. С точки зрения восприятия человеком объект можно определить одним из следующих способов:

- * Осязаемый и (или) видимый предмет.
- * Нечто, воспринимаемое мышлением.
- * Нечто, на что направлено мышление или действие.

Таким образом, мы расширили неформальное определение объекта, привязав его к окружающей действительности и определив его существование во времени и пространстве. Термин «объект» в программном обеспечении впервые введен в языке Simula и означал какой-либо аспект моделируемой реальности [2].

Объектами реального мира являются не только те объекты, которые интересны с точки зрения проектирования программных систем. Для отражения механизмов поведения объектов на верхнем уровне абстракции в процессе проектирования программы вводятся специальные виды объектов [3]. Это приводит нас к более четкому определению, данному Смиттом и Токем: «Объект представляет собой особый опознаваемый предмет, блок или сущность (реальную или абстрактную), имеющую важное функциональное назначение в данной предметной области» [4]. В еще более общем плане объект может быть определен как нечто, имеющее четко очерченные границы [5].

Представим себе завод, на котором создаются композитные материалы для таких разных целей, как велосипедные рамы и крылья самолетов. Заво-

ды часто разделяются на цеха: механический, химический, электрический и т.д. Цеха подразделяются на участки, на каждом из которых несколько единиц оборудования, таких, как штампы, прессы, станки. На производственных линиях можно увидеть множество емкостей с исходными материалами, из которых с помощью химических процессов создаются блоки композитных материалов, являющихся в свою очередь основой для конечного продукта — велосипедных рам или крыльев самолета. Каждый осязаемый предмет может рассматриваться как объект. Токарный станок имеет четко очерченные границы, которые отделяют его от обрабатываемого на этом станке композитного блока; рама велосипеда в свою очередь имеет четкие границы по отношению к участку с оборудованием.

Существуют такие объекты, для которых определены явные границы, но сами объекты представляют собой несвязанные события или процессы. Например, химический процесс на заводе: его границы явно определены взаимодействием компонентов, которые в течение определенного времени ведут себя известным образом. Аналогично система CAD/CAM позволяет определять линию пересечения сферы и куба. Хотя эта линия не существует отдельно от сферы и куба, она остается самостоятельным объектом в системе и ее границы четко определены.

Объект обладает состоянием, проявляет четко выраженное поведение и индивидуальность. Объекты могут быть осязаемыми, но иметь размытые физические границы. Например, реки, туман или толпы людей. Подобно тому, как взявший в руки молоток начинает видеть во всем окружающем только объекты для забивания, проскитываясь с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов. Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие, как время, красота, цвет, эмоция (например, любовь или гнев). Однако все перечисленное является свойствами, которые присущи объектам. Можно, например, утверждать, что некоторый человек (объект) любит свою жену (другой объект), или определенный кот (еще один объект) имеет серую шерсть.

Таким образом, требование иметь четкие границы не является достаточным, чтобы отделить один объект от другого или дать оценку качества абстракции. На основе имеющегося опыта можно дать следующее определение:

Объект обладает состоянием, поведением и индивидуальностью; структура и поведение схожих объектов определяют общий для них класс; термины «экземпляр класса» и «объект» — взаимозаменяемы.

Состояние

Понятие «состояние». Рассмотрим, например, торговый автомат, разливающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат «выдает» выбранный напиток. Что произойдет, если сначала будет нажата кнопка выбора напитка, а потом уже опущена монета? Большинство автоматов при этом никак не реагируют, так как действия человека не соответствуют заложенным в них правилам. Предположим, что пользователь автомата не обратил внимание на предупреждающий сигнал «Сделайте правильный выбор» и опустил в автомат еще одну монету. В большинстве случаев автоматы спокойно «проглатывают» такие монеты.

Итак, поведение объекта определяется последовательностью совершаемых над объектом действий. Такая зависимость поведения во времени объясняет-

ся наличием данных о состоянии объекта в его структуре. Для торгового автомата, например, существенными являются данные о количестве опущенных монет до нажатия кнопки выбора напитка. Кроме того, важно знать об имеющемся выборе напитков и их количестве.

На основе этого примера дадим следующее определение:

Состояние объекта характеризуется перечнем всех возможных (обычно статических) свойств данного объекта и текущими значениями (обычно динамическими) каждого из этих свойств.

Одним из свойств торгового автомата является также способность принимать монеты. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет, от которого зависит действие автомата. Число монет увеличивается по мере их опускания в автомат и уменьшается во время обслуживания при извлечении монет. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер автомата), поэтому в данном определении использованы термины «обычно динамические».

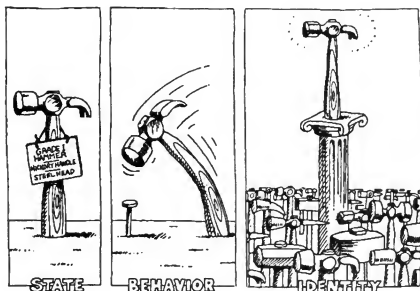
К числу свойств объекта относятся присущие ему или приобретаемые характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для подъемника характерным является то, что он сконструирован для подъема и спуска, но не для горизонтального перемещения. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу природы объекта. Выражение «как правило» означает, что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может определить некоторое препятствие как статическое, а затем определить, что это дверь, которую можно открывать. В этой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель объекта.

Все свойства объекта характеризуются значениями их параметров. Эти значения могут быть простыми количественными характеристиками, а могут означать другой объект. Состояние подъемника может описываться числом 3, означающим номер этажа, на котором подъемник в данный момент находится. Для торгового автомата состояние объекта включает множество других объектов, например имеющиеся в наличии смеси для приготовления напитков. Эти смеси являются самостоятельными объектами независимо от торгового автомата (так как могут существовать самостоятельно) и над ними можно совершать определенные действия.

Таким образом, мы установили различие между объектами и простыми величинами: простые количественные характеристики (например, числа) являются «постоянными, неизменными и неприходящими», тогда как объекты «существуют во времени, изменяются, имеют внутреннее состояние, преходящи и могут создаваться, разрушаться и разделяться» [6].

Тот факт, что всякий объект характеризуется состоянием, означает, что он занимает определенное пространство (физически или в памяти компьютера).

Примеры состояния. Предположим, что на языке C++ нам нужно создать регистрационные записи о персонале. Можно сделать это следующим образом:



Объект обладает состоянием, проявляет четко выраженное поведение и индивидуальность.

```
struct PersonnelRecord
```

```
{
    char          *name[100];
    int           socialSecurityNumber;
    char          *department[10];
    float         salary;
};
```

Каждая компонента в приведенной структуре обозначает конкретное абстрактное свойство. Объявление определяет не объект, а класс, поскольку оно не отражает какой-либо реальности. Для того чтобы создать объекты данного класса, необходимо дать следующее:

```
PersonnelRecord deb, dave, karen, jim, tom, denise;
```

В данном случае объявлено шесть различных объектов, каждый из которых занимает определенный участок в памяти. Хотя свойства этих объектов являются общими (их состояние представляется единообразно), в памяти объекты не пересекаются и занимают каждый свое место. На практике принято ограничивать доступ к элементам состояния объекта, а не делать их общедоступными, как в предыдущем определении класса. С учетом сказанного изменим данное определение следующим образом:

```
class PersonnelRecord {
public:
    char          *employeeName() const;
```



```

    int          employeeSocialSecurityNumber () const;
    char         *employeeDepartment () const;
protected:
    void         setEmployeeName (char *name);
    void         setEmployeeSocialSecurityNumber (int number);
    void         setEmployeeDepartment (char *department);
    void         setEmployeeSalary (float salary);
    float        employeeSalary () const;
private:
    char         *name[100];
    int          socialSecurityNumber;
    char         *department[10];
    float        salary;
};

```

Новое определение сложнее, и во многих отношениях содержательнее прежнего. В частности, в новом определении осуществлена защита для доступа к данным со стороны других объектов. Если структура этого класса будет в дальнейшем изменена, код придется перескомпилировать, но семантически другие объекты не будут зависимы от этих изменений (т.е. их код сохранится). Кроме того, решается также проблема занимаемой объектом памяти за счет явного определения операций над объектами данного класса. В частности, все объекты-пользователи могут узнать имя, номер страховки, место работы, но только ограниченному кругу объектов разрешено устанавливать значения указанных параметров. Не для всех объектов разрешен доступ к параметру, характеризующему заработную плату. Другое достоинство последнего определения связано с возможностью его переопределения. В предыдущем разделе уже было показано, что механизм наследования позволяет переопределить абстракцию и специализировать ее содержание.

В заключение скажем, что внутри каждого объекта хранятся в защищенном виде элементы, отражающие его состояние, и состояние всей системы в целом распределено между объектами.

Поведение

Понятие «поведение». Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение характеризует то, как объект воздействует или подвергается воздействию других объектов с точки зрения изменения состояния этих объектов и передачи сообщений.

Другими словами, «поведение объекта полностью определяется его действиями» [7].

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, объект-пользователь может активизировать операции Add и Pop для того, чтобы дать очередному объекту приращение или сократить его. Существует также операция Length_Of, которая позволяет определить размер объекта, но не может изменить значение этого размера. Применительно к таким языкам программирования, как Smalltalk, принято говорить о передаче сообщений между объектами. В основном понятие «сообщение» совпадает с понятием «операции»

над объектами, но механизм их действий различен. В дальнейшем эти два термина используются как синонимы.

Как правило, в объектных и объектно-ориентированных языках операции, выполняемые над данным объектом, называются *методами* (методической частью объекта) и входят составной частью в определение класса. В языке C++ для этих целей используется термин «функция-элемент».

Примеры поведения. На языке Ada опишем класс, определяющий очередь:

```
generic
  type Item is private;
package Simple_Queue is

  type Queue is limited private;
  procedure Copy      (From_The_Queue : in Queue;
                       To_The_Queue   : in out Queue);
  procedure Clear     (The_Queue      : in out Queue);
  procedure Add       (The_Item       : in Item;
                       To_The_Queue   : in out Queue);
  procedure Pop       (The_Queue      : in out Queue);
  function Is_Equal   (Left           : in Queue;
                       Right          : in Queue) return Boolean;
  function Length_Of  (The_Queue      : in Queue) return Natural;
  function Is_Empty   (The_Queue      : in Queue) return Boolean;
  function Front_Of   (The_Queue      : in Queue) return Item;

  generic
    with procedure Process (The_Item : in Item;
                           Continue : out Boolean);
  procedure Iterate      (Over_The_Queue : in Queue);
  Overflow               : exception;
  Underflow              : exception;

private
  ...
end Simple_Queue;
```

Приведенный тип Queue означает класс, но не объект. Поскольку мы имеем дело с обобщением (пакет типа generic), то класс будет параметризованным. Чтобы определить объекты очереди (целые значения), необходимо определить параметры в данном обобщенном пакете:

```
package Integer_Queue is new Simple_Queue (Item=>Integer);
```

Затем определим четыре объекта данной очереди:

```
A, B, C, D : Integer_Queue.Queue;
use Integer_Queue;
```

Теперь этими объектами можно управлять:

```
Add(1, To_The_Queue=>A);
Add(3, To_The_Queue=>A);
Add(5, To_The_Queue=>A);
copy (From_The_Queue=>A, To_The_Queue=>B);
Pop(B);
Pop(B);
```

Выполнение указанных операторов приведет к тому, что очередь А будет состоять из трех компонент (начиная с номера 1), а очередь В — из одной компоненты (начиная с номера 3). Каждый из объектов имеет при этом свое состояние, определяющее их поведение.

Понятие «операция». Из практики известно пять основных видов операций над объектами. Ниже приведены наиболее распространенные операции:

- * Модификатор Операция, которая изменяет состояние объекта путем записи или доступа
- * Селектор Операция, дающая доступ для определения состояния объекта без его изменения (операция чтения)
- * Итератор Операция доступа к содержанию объекта по частям (в определенной последовательности)¹⁾

Поскольку логика этих операций весьма различна, считается полезным при кодировании отметить эти различия. Так, при описании пакета Simple-Queue вначале определены все модификаторы (в виде процедур Copy, Clear, Add и Pop), затем все селекторы (в виде функций Is_Equal, Length_of, Is_Empty и Front_of) и в последнюю очередь итераторы (обобщенная процедура Iterate).

В языках Smalltalk, C++ и CLOS определяются еще два вида операций:

- * Конструктор Операция создания и (или) инициализация объекта
- * Деструктор Операция разрушения объекта и (или) освобождение занимаемой им памяти

В языке C++ конструктор и деструктор составляют часть описания класса, тогда как в Smalltalk и CLOS эти операторы определены в протоколе метакласса (т.е. класса классов).

В языке Smalltalk операции могут быть только методами, так как процедуры и функции вне классов в этом языке определять не допускается. Напротив, в языках Object Pascal, C++, CLOS и Ada допускается описывать операции как независимые от объектов подпрограммы. Такие общедоступные процедуры и функции служат в качестве операций над объектами или объектами одного или разных классов. Общедоступные процедуры группируются по отношению к классам, для которых они создаются. Это дает основание называть такие пакеты процедур утилитами класса. Например, на основе выше определенного пакета Integer_Queue можно дать описание следующей процедуры в отдельном пакете утилит класса:

```
procedure pop_Until_Item_Found      (The_Queue      : in out Integer_Queue.Queue;
                                     The_Item        : in Integer);
```

```
Item_Not_Found : exception;
```

¹⁾ Липман дал несколько иную классификацию: функции управления, функции реализации, вспомогательные функции и функции доступа [8].

Цель такой операции состоит в последовательном исключении элементов очереди до тех пор, пока не будет обнаружен заданный элемент. Если очередь окажется пустой до того, как будет обнаружен заданный элемент, то имеет место особая ситуация. Данная операция не является примитивной, она включает операции, входящие в класс `Simple_Queue`.

Таким образом, любой метод является операцией, но не любая операция является методом, так как могут иметь место общедоступные (свободные) процедуры. Практически большая часть операций определяется как методическая часть классов, хотя, как показано в следующем разделе, иногда целесообразно сделать иначе. Например, когда создается операция, воздействующая на несколько объектов разных классов, нет оснований относить такую операцию ни к одному из классов.

Совокупность всех методов и общедоступных процедур, относящихся к конкретному объекту, образует протокол этого объекта. Протокол объекта, таким образом, определяет оболочку поведения объекта, охватывающую его внутреннее статическое и внешнее динамическое проявления.

Объекты как автоматы. Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение. Это позволяет представить объект в качестве сложного независимого автомата [9]. Для ряда объектов такой временной порядок настолько существен, что наилучшим способом описания такого объекта является теория конечных автоматов.

Объекты могут быть активными и пассивными. Активным является тот объект, который расположен в канале управления. Активный объект в общем случае автономен, т.е. он может реализовать свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы составляют канал управления. Если система имеет несколько каналов управления, то и активных объектов может быть множество. В последовательных системах в каждый момент времени существует только один активный объект (и соответственно только один канал управления).

Индивидуальность

Понятие «индивидуальность». Кхошафиан и Копеланд предложили следующее определение:

«Индивидуальность — это такие свойства объекта, которые отличают его от всех других объектов» [10].

Они отмечают, что «в большинстве языков программирования и баз данных для различения временных объектов, их взаимной адресации и идентификации используются имена переменных» [11]. Источником множества ошибок в объектно-ориентированном программировании является невозможность отличить имя объекта от самого объекта.

Примеры индивидуальности. Сделаем следующее определение на языке `Object Pascal`:

```
TPaletteItem = object (TObject)
```

```
  kId       : Integer;
  kFrame    : Rect;
```

```
  procedure TPaletteItem.Initialize (ID : Integer; Frame : Rect);
  procedure TPaletteItem.Free; override;
```

```

procedure TPaletteItem.Draw (Area : Rect);
procedure TPaletteItem.Highlight (FromHL, ToHL : HLState);
function TPaletteItem.IsMouseHit (TheMouse : Point) : Boolean;
end;
```

TPaletteItem является определением класса и должно быть приведено в разделе описания типов программного блока.

Объявить объекты (экземпляры) такого класса можно следующим образом:

```
Item_1, Item_2, Item_3 : TPaletteItem;
```

Это объявление должно быть сделано в разделе переменных.

В результате такого определения (рис. 3-1, а) в памяти резервируются три области (называемые в Object Pascal переменными-ссылками на объект) с именами Item_1, Item_2, Item_3, начальное значение которых не определено, т.е. в данный момент эти имена не связаны ни с какими объектами. Для создания самих объектов необходимо в явном виде использовать процедуру new:

```
new(Item_1); new (Item_2);
```

Выполнение каждой процедуры создает отдельный объект и связывает этот объект с переменной-ссылкой. Поскольку указанные процедуры не инициализируют состояние объекта, это нужно сделать специально:

```

SetRect (ARect, 1, 1, 32, 32, );
Item_1.Initialize (1001, ARect);
SetRect (ARect, 1, 33, 32, 64, );
Item_2.Initialize (1002, ARect);
```

Результат выполнения приведенных выше операций показан на рис. 3—1, б. Свойство индивидуальности сохраняется даже при полном изменении состояния. Это напоминает вопрос Зенона о том, остается ли река сама собой в процессе времени, хотя никогда уже прежняя вода не потечет по ней? Для примера выполним теперь следующие операторы:

```

Item_2.kld :=1003;
new (Item_3);
SetRect (ARect, 1, 33, 32, 64);
Item_3.Initialize (1003, ARect);
```

Объект, обозначенный Item_2, по-прежнему существует, хотя его состояние изменилось. Был создан новый объект с именем Item_3, отличный от Item_2, хотя их состояния одинаковы. Отметим, что выражение «объект, обозначенный именем Item_2», будет более верным, чем выражение «объект Item_2», но в дальнейшем двумя этими фразами будем обозначать одно и то же. Что произойдет, если выполнить следующий оператор:

```
Item_1 := Item_2;
```

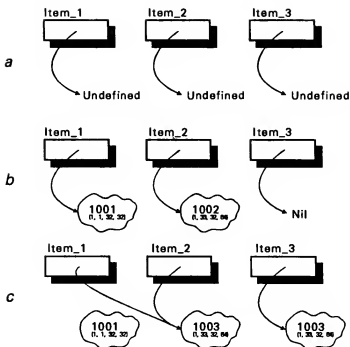


Рис. 3-1. Пояснение понятия «индивидуальность объектов».

Оказывается, что результат будет нежелательным, как видно из рис. 3-1, b. Во-первых, объект обозначенный ранее Item_1, становится теперь недоступен; он навсегда потерян и превращен в обычный «мусор». Имена Item_1 и Item_2 теперь обозначают один и тот же объект. Такая ситуация называется структурной неопределенностью и означает, что один объект имеет несколько имен (раздвоение). Структурная неопределенность может иметь опасные последствия, так как позволяет изменить состояние объекта через первое имя, скрыв этот факт от объектов, пользующихся другим именем. Выполним следующий оператор:

```
Item_1.Id:=1004;
```

Мы видим, что одновременно изменилось состояние объекта, обозначенного Item_2.

Операции присваивания и тождество объектов. Структурная неопределенность возникает из-за дублирования индивидуальности объектов путем присвоения им нового имени. В большинстве случаев такая двойственность просто недопустима. Поэтому очень важно уметь определять такие ситуации

и семантически их предотвращать. В используемых нами языках для предотвращения указанных ситуаций используются следующие операции присвоения:

* Smalltalk	<=
* Object Pascal	:=
* C++	=
* CLOS	let
* Ada	:=

Чтобы скопировать объект и избежать структурой неопределенности, в языке Smalltalk введены методы shallowCopy (копирующий только объект) и deepCopy (копирующий объект, включая его состояние). В языке Object Pascal той же цели служат предопределенные методы Clone и ShallowClone. В языке C++ эту же роль выполняет оператор присвоения (который переопределяется с целью обеспечения требуемой семантики). В языке CLOS и Ada необходимо явно определить операторы копирования для каждого класса.

С вопросом присвоения тесно связан вопрос тождественности. Тождественность представляется достаточно простой концепцией, но может обозначать одну из двух вещей. Во-первых, тождественность можно понимать как использование двух имен для обозначения одного и того же объекта. Во-вторых, тождественность может обозначать наличие одинакового состояния у двух разных объектов. В примере на рис. 3-1, b оба варианта тождественности будут справедливы для Item_1 и Item_2. Однако для Item_2 и Item_3 истинным будет только второй вариант.

В разных языках по-разному обозначается также условие тождественности. Для проверки того, относятся ли два имени к одному объекту, используются следующие операторы:

* Smalltalk	= =
* Object Pascal	=
* C++	= =
* CLOS	eql
* Ada	=

В Smalltalk и C++ эти операторы могут быть переопределены. Следующие операторы используются для проверки эквивалентности состояний двух объектов:

* Smalltalk	=
* Object Pascal	user defined
* C++	user defined
* CLOS	equalp
* Ada	user defined

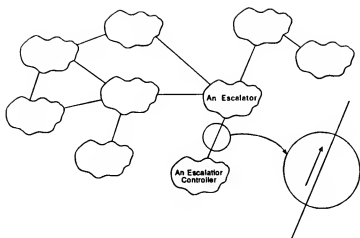


Рис. 3-2. Отношение использования между объектами.

Время существования объектов. Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием — момент изъятия отведенного участка памяти. Следующие методы используются для создания объектов:

* Smalltalk	new
* Object Pascal	new
* C++	new
* CLOS	make-instance
* Ada	new

В языке C++ создание объекта автоматически приводит к вызову конструктора для данного класса объектов. Smalltalk, C++ и CLOS позволяют переопределить методы создания объектов. В C++ и Ada для создания объектов в свободном участке памяти (динамической области) применяется процедура new. В этих двух языках возможно также создание временного объекта прямо в программном стеке путем объявления переменной данного класса.

Объект продолжает существовать до тех пор, пока он занимает место в памяти, даже если будет потеряна ссылка на этот объект. В языках Smalltalk и CLOS определены процедуры «сборки мусора», в процессе которых автоматически разрушаются все объекты, лишенные ссылок на них. В языках C++ и Ada объекты, созданные в программном стеке, ликвидируются автоматически при переходе управления за пределы области действия переменной, связанной с данным объектом. Следующие методы используются для явного разрушения объектов:

* Smalltalk	release
* Object Pascal	free
* C++	delete
* Ada	delete

В C++ при ликвидации объекта автоматически вызывается деструктор соответствующего класса. В C++ и Ada к объектам созданным в динамической области памяти может быть применен метод delete. Переопределение деструкторов допускается в языке Smalltalk и C++.

3.2. ОТНОШЕНИЯ МЕЖДУ ОБЪЕКТАМИ

Типы отношений между объектами

Сами по себе объекты не представляют никакого интереса, только в процессе взаимодействия объектов между собой реализуется цель системы. По выражению Ингалса: «Вместо бессистемной кусочной обработки структур данных мы получаем объекты с ясным поведением, которые обращаются друг с другом по тщательно проработанному интерфейсу и выполняют нужные действия» [12]. Рассмотрим, например, структуру самолета, которая определяется как «совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевает эту тенденцию» [13]. Только за счет согласованных усилий всех компонентов самолета он имеет возможность летать.

Отношения двух любых объектов основываются на предположении, что каждый объект имеет информацию о другом объекте, об операциях, которые над ним можно выполнять, и об ожидаемом поведении. Особый интерес для OOD представляют два типа иерархических соотношений объектов:

- * Отношение использования.
- * Отношение включения.

Зейдевич и Старк назвали эти два типа отношений отношениями старшинства и родства соответственно [14].

Отношение использования

Отношение использования между объектами. Отношения использования между несколькими объектами проиллюстрированы на рис. 3-2. На этом рисунке линия между двумя условными объектами обозначает наличие отношений использования между ними и подразумевает возможность передачи сообщений по этой линии. На выносной части рисунка (увеличено) видно, что при использовании объектом AnEscalator Controller объекта AnEscalator первый посылает второму сообщение (отмечено стрелкой вдоль линии).

Пересылка сообщений между объектами обычно односторонняя, но возможны и двусторонние связи. Каждый объект, включенный в отношения использования, может выполнять следующие три роли:

- * Воздействие Объект может воздействовать на другие объекты, но сам никогда не подвержен воздействию других объектов; в определенном смысле соответствует понятию активный объект

- * **Исполнение** Объект в этом случае может только подвергаться управлению со стороны других объектов, но никогда не выступает в роли воздействующего объекта
- * **Посредничество** Такой объект может выступать как в роли воздействующего, так и в роли исполнителя; как правило, объект посредник создается для выполнения операций в интересах какого-либо активного объекта или другого посредника

Примеры отношений использования. Рассмотрим, например, химический процесс, где имеются два клапана для выпуска двух различных жидкостей в реактор и один выпускной клапан. Для протекания реакции необходимо плавно повысить температуру веществ до определенной величины, затем поддерживать ее в течение определенного времени, после чего охладить до комнатной температуры. Одной из ключевых абстракций в такой задаче является нагреватель, класс которого определим на языке CLOS следующим образом:

```
(defclass heater () ())
(defmethod turn_on ((h heater))...)
(defmethod turn_off ((h heater))...)
(defmethod current_temperature ((h heater))...)
```

Здесь дано определение класса `heater` и трех его методов. Реализация методов дана в краткой записи, поскольку подробности в данном случае несущественны. Для образования экземпляра данного класса необходимо дать следующее описание:

```
(setq a_heater (make-instance 'heater))
```

Затем можно определить класс реактора (`crucible`):

```
(defclass crucible ()
  ((temperature :initform 0 :accessor set_point)))
((defmethod set_temperature ((c crucible) (f float) (s integer))
  (turn_on a_heater)
  ...
  (turn_off a_heater))
```

Реализация метода `set_temperature` частично опущена для краткости. Этот метод имеет своей целью довести реактор с до температуры `f` за `s` секунд. Для достижения этой цели используются послышки сообщений на включение и отключение нагревателя (`turn_on` и `turn_off` соответственно). Для создания экземпляра объекта `crucible` необходимо следующее описание:

```
(setq a_crucible (make-instance 'crucible))
```

Теперь мы определим класс клапана (valve) и создадим три объекта этого класса:

```
(defclass valve () ())
(setf v_1 (make-instance 'valve))
(setf v_2 (make-instance 'valve))
(setf v_3 (make-instance 'valve))
```

Будем полагать, что методы, позволяющие открывать и закрывать клапаны, определены. Наконец, необходимо ввести класс, соответствующий контроллеру процесса (process_controller), управляющему работой клапанов и реактора, и создать экземпляр объекта данного класса:

```
(defclass process_controller () ())
(defmethod start_process ((c process_controller))...)
(setf a_process_controller (make-instance 'process_controller))
```

Посмотрим, что произойдет при следующем вызове метода:

```
(start_process a_process_controller)
```

Во-первых, будут сообщения от объекта с именем a_process_controller к объектам с именами value_1 и value_2 (для открытия и последующего закрытия этих двух клапанов), после чего будет передано сообщение set_temperature объекту a_crucible. Объект с именем a_crucible в свою очередь формирует сообщение turn_on и turn_off для отправки объекту a_heater. Наконец, объект a_process_controller передает сообщение объекту value_3 для освобождения реактора от продукта.

В приведенном примере объекты a_heater, value_1, value_2 и value_3 являются исполнителями: они получают команды от других объектов, а сами на другие объекты не воздействуют. Объекты a_crucible и a_process_controller являются посредниками, поскольку они подвергаются воздействию других объектов и сами являются воздействующими.

Понятие синхронизации. При передаче сообщений от одного объекта к другому оба взаимодействующих объекта должны определенным образом синхронизироваться. Для последовательных систем такая синхронизация, как правило, реализуется через вызовы подпрограмм. Однако в параллельных системах ситуация гораздо сложнее, поскольку для многоканального управления необходимо решить проблему исключительных ситуаций. Отсюда вытекает еще один способ классификации объектов:

- * Объект-транслятор Пассивный объект, имеющий только один канал управления
- * Блокированный объект Пассивный объект, имеющий несколько каналов управления
- * Параллельный объект Активный объект, имеющий несколько каналов управления

Ниже в настоящей главе приводятся примеры только транслирующих объектов.

Отношение включения

Понятие отношения включения между объектами. Логично предположить, что конкретный объект-подъемник состоит из других объектов, например из мотора и датчика перемещений. Другими словами, последние два объекта являются элементами состояния объекта-подъемника, как показано на рис. 3-3.

Между отношениями включения и использования существует взаимная связь. Включение одних объектов в другие предпочтительнее в том плане, что при этом уменьшается число объектов, с которыми приходится оперировать на данном уровне описания. С другой стороны, использование одних объектов другими имеет преимущество: не возникает сильной зависимости между объектами, как в случае включения. В процессе проектирования необходимо тщательно взвешивать оба указанных фактора.

Примеры отношений включения. При построении абстракций квартиры, очевидно, необходимо включить в нее кухню, ванную комнату, спальню и гостиную. На языке Smalltalk это записывается следующим образом:

```
Object subclass: #Apartment
instanceVariableNames: 'aKitchen aBathroom aBedroom aFamilyRoom'
classVariableNames: ' '
poolDictionaries: ' '
category: 'Simulation'
```

При образовании каждого экземпляра объекта класса Apartment образуются и все составляющие его объекты-компоненты. Предполагая, что классы, соответствующие кухне, ванной, спальне и гостиной, уже определены, мы определим следующий метод, входящий в класс Apartment:

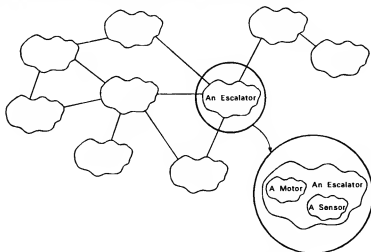


Рис. 3-3. Отношение включения между объектами.

initialize

•Initialize the apartment by creating its rooms.»

aKitchen <— Kitchen new.

aBathroom <— Bathroom new.

aBedroom <— Bedroom new.

FamilyRoom <— FamilyRoom new.

Теперь приведем оператор, создающий и инициализирующий локальный объект класса Apartment:

! anApartment !

anApartment <— Apartment new Initialize.

В качестве побочного эффекта процедуры «инициализация» образуются все четыре объекта. По отношению к такому объекту, как Apartment, применяются термины *сложный*, *составной* или *агрегатированный*.

3.3. СУЩНОСТЬ «КЛАСС»

Что такое класс?

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие в этих двух понятиях. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию, «выжимку» из объекта. Таким образом, можно говорить о классе «Млекопитающие», который включает общие характеристики для всех млекопитающих. Для указания на конкретного представителя млекопитающих необходимо сказать «это млекопитающее» или «то млекопитающее».

В общепотребительных терминах можно дать следующее определение класса: «группа, множество или вид с общими свойствами или общим свойством, разновидностями, отличиями по качеству, возможностями или условиями» [15]. С точки зрения OOD дадим следующее определение класса:

Класс — множество объектов, связанных общностью структуры и поведения.

Любой объект является просто экземпляром класса.

Что не является классом? Объект не является классом, хотя в дальнейшем мы увидим, что класс может быть объектом. Объекты, не связанные общностью структуры и поведения, не могут образовать класс, так как по определению они не связаны между собой ничем, кроме их общей природы как объектов.

Внешние и внутренние проявления класса

Мейер [16] и Снайдер (Snyder) [17] высказали точку зрения о том, что программирование в основном — это вопрос «взаимодействия»: большая задача и составляющие ее функции разделяются на более мелкие с одновременным определением взаимодействия этих компонентов в системе. Наиболее очевидно эта идея обнаруживается в проектировании классов.

Класс служит для представления совокупности объектов общей структуры и общего поведения. В то время как отдельный объект существует коик-

ротно и играет в системе определенную роль, класс содержит описание структуры и поведения всех объектов, связанных отношением общности. Таким образом, класс выполняет роль своего рода соглашения о связях в отношении абстракции и всех ее реализаций. В языках со строгой типизацией имеется возможность выявления нарушений такого соглашения о связях во время компиляции.

Это достигается за счет того, что в интерфейсной части описания класса содержатся необходимые данные о проектных решениях. Приняв такую точку зрения на программирование как на процесс контрактации, мы приходим к явному разделению внутреннего и внешнего проявления класса. Интерфейсная часть описания класса соответствует его внешнему проявлению, подчеркивает его абстрактность, но скрывает структуру и особенности поведения. В первую очередь интерфейсная часть класса состоит из перечня действий, который допускает описание других классов, констант, переменных и особенностей, необходимых для полного определения данной абстракции. Реализация класса, напротив, составляет его внутреннее проявление и определяет особенности поведения. В этой части раскрывается реализация тех операций, которые перечислены в интерфейсной части описания.

Интерфейсная часть описания класса может быть разделена на три составные части:

- * Общедоступная Та часть интерфейса класса, в которой даются определения, «видимые» для всех объектов-пользователей данного класса
- * Защищенная Та часть интерфейса класса, в которой даются определения, «видимые» только для объектов, относящихся к подклассам данного класса
- * Обособленная Та часть интерфейса класса, в которой даются определения, «скрытые» для объектов всех других классов

Полностью такое разделение интерфейса класса реализуется только в языке C++ (из числа языков, упоминаемых в данной книге). При необходимости возможно определение на языке C++ структур без ограничения доступа. В Ada допускается выделение общедоступной и обособленной частей, но отсутствует описание защищенной части. В языках Smalltalk, Object Pascal и CLOS разделение интерфейса на части должно реализовываться путем программных соглашений.

Структура состояния объекта также требует определенного описания, которое, как правило, состоит в объявлении констант и переменных в обособленной части описания интерфейса. Это делает общую часть структуры объектов закрытой для доступа, и, следовательно, внесение в эту часть изменений не отражается на функционировании объектов-пользователей. В языке Smalltalk указанная часть определений может быть только обособленной. В языках C++, CLOS и Ada допускаются такие определения как в обособленной, так и общедоступной части интерфейса класса, а в Object Pascal структура класса всегда общедоступна.

У внимательного читателя может возникнуть вопрос: почему структура объекта является частью интерфейса класса (пусть даже и обособленная), а

не входит в реализацию? Такое решение имеет чисто практический интерес: другое решение требует либо создания специальных аппаратных средств, либо значительно усложнит компилятор. В частности, при обработке компилятором следующего определения на C++:

```
Shape aShape;
```

требуется точно знать необходимый размер памяти, отводимый для этого объекта. Если структура класса определена вне его интерфейса, необходимо точно определить реализацию класса для того, чтобы точно пользователи могли оперировать таким объектом. Это в принципе нарушает идею разделения класса на внешнюю и внутреннюю части.

Константы и переменные, составляющие структуру класса в различных языках, обозначаются разными терминами. В языке Smalltalk используется термин «переменная объекта», в языке Object Pascal — «поле», в C++ — «фрагмент объекта», а в CLOS ... термин «слот» (соответственно instance variable, field, member object, slot). В дальнейшем эти термины мы будем использовать как синонимы и обозначать структуру состояния объекта.

3.4. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

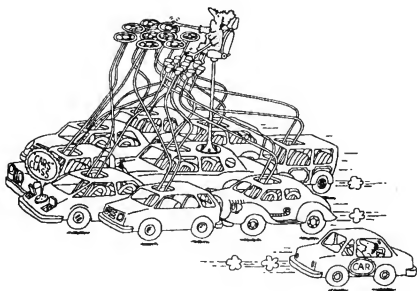
Типы отношений

Рассмотрим сходства и различия классов для следующих объектов: цветы, маргаритки, красивые розы, желтые розы и лепестки. Сделаем следующие выводы:

- * Маргаритка — вид цветка.
- * Роза — (другой) вид цветка.
- * Красная и желтая розы — разновидности розы.
- * Лепесток является частью обоих видов цветов.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. Наоборот, структура классов для конкретной области формируется на основе ключевых абстракций этой области и их связей [18]. Отношение между двумя классами следует рассмотреть по двум причинам. Во-первых, отношения классов могут указывать на какой-либо вид общности. Например, маргаритки и розы являются разновидностями цветов, имеют яркую окраску лепестков, сильный аромат и т.д. Во-вторых, отношения классов могут влиять на семантику связи между ними. Можно отметить, что между красными и желтыми розами больше сходства, чем между розами и маргаритками, а между маргаритками и розами больше, чем между лепестками и цветами.

Известно три основных типа отношений между классами [19]. Первый тип называется отношением «разновидность» и отражает степень общности. Например, фраза «роза является разновидностью цветов» означает, что роза является более специализированным подклассом класса цветов. Второй тип отражает агрегатирование объектов и называется отношением «составная часть». Так, например, лепесток — не разновидность цветка, а его составная часть. Третий тип обозначает отношение ассоциативности, т.е. смысловую связь между классами, которые не связаны никакими другими типами отношений. Примером могут служить два достаточно независимых класса роз и маргариток, которые соответствуют объектам, пригодным для декоративного оформления обеденного стола.



Класс обозначает множество объектов, имеющих общую структуру и общее поведение.

Языки программирования реализуют несколько общих способов для отражения трех типов отношений между классами. В частности, объектные и объектно-ориентированные языки программирования реализуют в разных комбинациях следующие механизмы отношения классов:

- * Наследование.
- * Использование.
- * Представление.
- * Метаклассы.

Особый подход к реализации отношений наследования называется делегированием (delegation), когда объекты рассматриваются в качестве прототипов (образцов), которые делегируют свое поведение другим объектам, ограничивая потребность в создании новых классов [20].

Отношения наследования составляют наиболее эффективный тип отношений и могут использоваться как для отражения общности, так и для отражения ассоциативности. На практике наследуемость не позволяет отразить все богатство отношений между ключевыми абстракциями данной предметной области. Для описания агрегатирования необходимо реализовать отношения использования. Далее следуют отношения представления, которые, подобно наследованию, охватывают обобщение и ассоциативность, но совершенно другим образом. Особым типом отношений являются метаклассы, которые реализуются далеко не всеми объектными и объектно-ориентированными языками программирования. Метакласс — это класс классов, позволяющий трактовать классы как объекты.

Отношение наследования

Примеры отношений наследования между классами. Находящиеся в полете космические зонды посылают на наземные станции информацию о состоянии своих основных систем (например, источников энергоснабжения и двигателей) и измерения датчиков (датчики радиации, масс-спектрометры, телекамеры, фиксаторы столкновений с микрометеоритами и т.д.). Вся совокупность передаваемой информации называется *телеметрическими данными*. Как правило, телеметрические данные передаются в виде пакетов, состоящих из заголовка (включающего временные метки и ключи для идентификации последующих данных) и нескольких фрагментов данных от подсистем и датчиков. Поскольку передаваемые данные строго упорядочены, запрашивается описание каждого вида таких данных в виде записей структурного типа. На языке C++ это выглядит так:

```
struct Time {
    int elapsedDays;
    int seconds;
};
struct ElectricalData {
    Time    timeStamp;
    int     id;
    float   fuelCell1Voltage, fuelCell2Voltage;
    float   fuelCell1Amperes, fuelCell2Amperes;
    float   currentPower;
};
```

Однако такое описание имеет ряд недостатков. Во-первых, структура `ElectricalData` не защищена, т.е. объект-пользователь может вызвать изменение такой важной информации, как элемент `id` или `current Power` (мощность, развиваемая двумя химическими батареями). Во-вторых, эта структура является полностью открытой, т.е. при ее модификации (добавлении новых элементов в структуру или изменении типа существующих элементов) нужно корректировать работу объектов-пользователей. Придется заново компилировать все описания, связанные каким-либо образом с этой структурой. Еще важнее, что внесение в эту структуру изменений может нарушить логику отношений с объектами-пользователями, а следовательно, логику всей программы. Кроме того, приведенное описание структуры очень трудно для восприятия. По отношению к такой структуре можно выполнить множество различных действий (пересылка данных, вычисление контрольной суммы для определения ошибок и т.д.), но все эти действия не будут связаны с приведенной структурой логически. В завершение предположим, что анализ требований к системе обусловил наличие нескольких сотен разновидностей телеметрических данных, включая другие электрические параметры, предшествующую информацию и замеры напряжения в разных контрольных точках системы. Очевидно, что описание такого количества дополнительных структур превысит приемлемый уровень как с точки зрения повторяемости структур, так и с точки зрения числа функций их обработки.

Подкласс может наследовать как структуру, так и поведение от своих суперклассов. Лучшим способом сохранения единства подхода к проекту является создание для каждого вида телеметрических данных отдельного класса, что позволит защитить данные в каждом классе и увязать их с выполняемыми операциями. Этот же подход решает проблему избыточности описания.

Еще лучше и последовательнее построить иерархию классов, в которой на основе более общих классов с помощью наследования образуются более специализированные; например, следующим образом:

```
class TelemetryData {
public:
    TelemetryData ();
    TelemetryData (const TelemetryData&);
    virtual ~TelemetryData ();
    virtual void send ();
    Time currentTime () const;
protected:
    int id;
private:
    Time timeStamp;
};
```

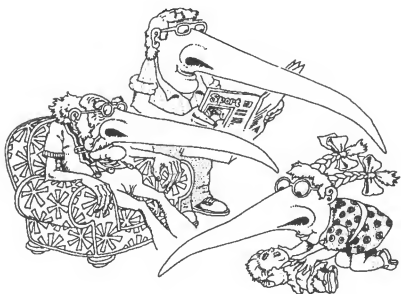
В этом примере введен класс, имеющий конструктор, деструктор и функции `send` и `currentTime`, видимые для всех объектов-пользователей.

Элемент `id` определен в защищенной части описания и видим только для класса `TelemetryData`. В то же время общедоступная функция `currentTime` позволяет всем объектам получить значение `timeStamp`, но изменить его при этом невозможно. Теперь перепишем класс `ElectricalData`:

```
class ElectricalData: public TelemetryData {
public:
    ElectricalData (float v1, float v2, float a1, float a2);
    ElectricalData (const ElectricalData&);
    virtual ~ ElectricalData ();
    virtual void send ();
    virtual float currentPower () const;
protected:
    float fuelCell1Voltage, fuelCell2Voltage, fuelCell1Amperes, fuelCell2Amperes;
};
```

Этот класс образован наследованием класса `TelemetryData`, но исходная структура дополнена (четырьмя новыми элементами), а поведение определено (изменена функция `send`). Почему значение `currentPower` не реализовано в виде элемента структуры, как в предыдущем описании `ElectricalData`? Это не является необходимым, так как данное значение легко вычисляется косвенным образом с помощью функции `currentPower`.

Понятие простого наследования классов. Определено, что наследование — такое отношение между классами, когда один класс повторяет структуру и поведение другого (*простое наследование*) или других (*множественное наследование*) классов. Класс, структура и поведение которого наследуются, называется *суперклассом*. Так, `TelemetryData` является суперклассом для `ElectricalData`. Производный от суперкласса класс называется *подклассом* по отношению к `TelemetryData`. Это означает, что наследование устанавливает между классами иерархию «по номенклатуре». В этом смысле `ElectricalData` является более специализированным классом от более общего `TelemetryData`. Мы уже видели, что в подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Наличие такого механизма отличает объектно-ориентированные языки проектирования от объектных.



Подкласс может наследовать структуру и поведение своего суперкласса.

Отношения простого наследования от суперкласса `TelemetryData` показаны на рис. 3-4. Стрелками на рисунке показаны отношения типа «разновидность» или «является». В частности, `CameraData` — это разновидность класса `SensorData`, который в свою очередь является разновидностью суперкласса `TelemetryData`. Такой же тип иерархии характерен для семантических сетей, которые часто используются специалистами по распознаванию образов и искусственному интеллекту для организации баз знаний [21]. В главе 4 показано, что правильная организация иерархии абстракций — это вопрос логической классификации.

Можно ожидать, что для некоторых классов на рис. 3-4 будут созданы экземпляры объектов, а для других не будут. Наиболее вероятно образование объектов из наиболее специализированных классов `ElectricalData` и `SpectrometerData`. Образование объектов из классов, занимающих промежуточное положение, более общее значение (`SensorData` и даже `TelemetryData`), менее вероятно. Такие классы, для которых не определены реализации объектов, называются абстрактными классами. На основе абстрактных классов образуются подклассы, дополненные в структурной и главной образам методической части. В языке Smalltalk допускается переопределение метода в подклассе за счет использования метода `SubclassResponsibility`. При невозможности переопределения вызов такого метода приводит к ошибке исполнения. В языке C++ метод абстрактного класса может быть заблокирован с помощью его инициализации в подклассе нулевым (пустым) значением. Такой метод называется *чистой виртуальной функцией*, а механизмы языка допускают создание объектов, экспортирующих такие функции.

Самый общий класс в структуре классов называется *базовым классом*. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие категории абстракций в конкретной предметной области. В некоторых языках программирования определен базовый класс самого верхнего уровня, который является единственным суперклассом для всех остальных классов.

В языке Smalltalk эту роль играет класс Object, а в Object Pascal — это TObject. В языке CLOS неявно определен класс standard-class в качестве единого суперкласса всех классов defclass; t является неявным суперклассом для standard-class и для большинства простых типов. В C++ общий базовый класс анонимен.

Для любого класса обычно определяются два вида пользователей [22]:

- * Экземпляры данного класса.
- * Производные подклассы.

Часто оказывается полезным различать интерфейс для этих двух разновидностей [23], чем объясняется наличие общедоступной, защищенной и обособленной части описания класса на языке C++: разработчик может четко разделить, какие элементы класса доступны для экземпляров, подклассов, для тех и других. В языке Smalltalk степень такого разделения меньше: элементы структуры «видимы» для подклассов, но не для экземпляров, а методическая часть общедоступна (допускается определять методы в качестве обособленных, но это не обеспечивает защиту). В Object Pascal поля и методы общедоступны, а в CLOS слоты доступны для подклассов, а для экземпляров «видимость» контролируется квалификаторами :reader, :writer и :accessor (чтение, запись, доступ соответственно).

Наследование подразумевает повторение структуры суперкласса. В предыдущем примере экземпляры класса ElectricalData содержат элементы структуры суперкласса (id и timeStamp) и элементы специализированного класса (fuelCell1Voltage, fuelCell2Voltage, fuelCell1Amperes, fuelCell2Amperes). Языки Smalltalk, Object Pascal, C++ и CLOS позволяют дополнять структуру суперкласса в подклассах. Однако в этих языках нельзя использовать для фрагмента данных уже существующее имя, так же как не допускается и сокращение структуры суперкласса в подклассах. Между наследованием и ограничением доступа существует реальное противоречие. В процессе наследования в значительной степени открывается строение суперкласса, т.е. для понимания строения конкретного класса нужно изучить все его суперклассы, включая в некоторой степени их внутреннее строение.

Поведение суперклассов также наследуется. Применительно к объектам класса ElectricalData можно использовать операции currentTime (унаследована от суперкласса), currentPower (определена в суперклассе) и send (переопределена по отношению к суперклассу). В большинстве языков допускается не только наследование методов суперкласса, но также исключение, добавление новых и переопределение существующих. В Smalltalk, Object Pascal и CLOS любой метод суперкласса можно переопределить в подклассе (в языке CLOS это называется обобщенной функцией). В Object Pascal для переопределения

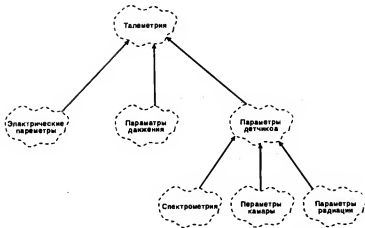


Рис. 3-4. Отношение простого наследования классов.

метода принято использовать ключевое слово `override` в определении подкласса. В C++ степень управления этим свойством несколько выше. Функция, объявленная виртуальной (функция `send` в предыдущем примере), может быть в подклассе переопределена, любая другая функция не может быть переопределена (как функция `currentTime`).

Понятие простого полиморфизма. Функция `send` для класса `TelemetryData` может быть реализована следующим образом:

```

void TelemetryData :: send () {
// transmit the id
// transmit the timeStamp
};
  
```

Заметим, что класс, к которому относится данная функция, назван явно. Тело этой функции является неполным (выполняемые действия определены в виде комментариев), но сам пример демонстрирует факт возможности отделения интерфейса метода от его реализации (в языке CLOS при объявлении метода неявно вводится обобщенная функция). Возвращаясь к реализации метода `send` в классе `ElectricalData`, напомним

```

void ElectricalData :: send () {
TelemetryData ::send ();
// transmit the fuelCell1Voltage and the fuelCell2Voltage
// transmit the fuelCell1Amperes and the fuelCell2Amperes
// transmit the currentPower
};
  
```

Такая реализация предусматривает выполнение операций, определенных в суперклассе, а затем дополнительной операции по пересылке данных соответствующих данному специализированному классу.

Определим теперь экземпляры двух описанных выше классов (хотя для класса `TelemetryData` это не типично):

```
TelemetryData telemetry;  
ElectricalData electrical (5.0, -5.0, 3.0, 7.0);
```

При наличии объектов, обозначенных именами `telemetry` и `electrical`, дадим следующее определение функции:

```
void sendTelemetryData (TelemetryData &D) {  
    D.send ();  
};
```

Что произойдет при выполнении двух следующих операторов?

```
sendTelemetryData (telemetry);  
sendTelemetryData (electrical);
```

В первом случае осуществляется пересылка данных `id` и `timeStamp`, а во втором кроме тех же данных пересылаются еще пять действительных чисел. Как это получается? Ведь тело функции `SendTelemetryData` состоит из единственного оператора `D.Send()`, который не отличается для класса `D`. Причина состоит в полиморфизме. Полиморфизм — это такой элемент теории типизации, который позволяет использовать одно имя (параметр `D`) для обозначения объектов различных классов, имеющих общий суперкласс.

В результате объект с таким именем может по-разному реагировать на выполнение общего набора операций. Карделли и Вегнер заметили, что «традиционные типизированные языки типа Pascal основаны на идее о том, что функции и процедуры, а следовательно, и операнды должны иметь определенный тип. Это свойство называется мономорфизмом т.е. каждая переменная и каждое значение относятся к одному определению типу. В противоположность мономорфизму полиморфизм допускает отнесение значений и переменных к нескольким типам» [24]. Впервые полиморфизм описал Страчи [25], который ввел особый вид полиморфизма, в котором символы, такие, как `+`, могут иметь различное значение. В настоящее время этот подход носит название «перегрузка» (*overloading*). Например, в языках C++ и Ada можно объявлять несколько процедур или функций, имеющих одинаковое имя, но различающихся перечислением параметров, количеством и типом аргументов и возвращаемых значений. Тем же автором введен термин «параметрический полиморфизм», который мы теперь называем просто полиморфизмом.

При отсутствии полиморфизма код программы вынужденно содержит множество операторов варианта или переключения. Например, на языке Pascal невозможно образовать иерархию классов телеметрических данных; вместо этого придется определить одну большую вариантную запись, включающую все разновидности данных. Для выбора варианта нужно проверить значение параметра, связанного с типом записи. На языке Pascal процедура `sendTelemetryData` может быть написана следующим образом:

```
const  
    Electrical      = 1;  
    Propulsion      = 2;  
    Spectrometer    = 3;  
...
```

```

procedure Send_Telemetry_Data (TheData: Data);
begin
    case TheData.Kind of
        Electrical : SendElectricalData (TheData);
        Propulsion : SendPropulsionData (TheData);
        ...
    end
end;

```

Чтобы ввести новый тип телеметрических данных, нужно модифицировать эту вариантную запись и расширить оператор варианта, который управляет выбором варианта. В такой ситуации увеличивается вероятность ошибок и проект становится нестабильным.

Наследование позволяет разделить различные разновидности абстракций, отказавшись от введения таких монолитных типов. Каплан и Джонсон отметили, что «полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют одинаковые протоколы» [26]. Полиморфизм позволяет обойтись без больших операторов варианта, поскольку сами объекты содержат сведения о типе данных. Возможна реализация наследования без полиморфизма, но эффективность такого механизма очень низка. Это видно на примере языка Ada, где можно объявить производные типы, но из-за мономорфизма языка уже в период компиляции действительный тип операций полностью определяется. Полиморфизм тесно связан с механизмом позднего связывания. При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В языке C++ программист имеет возможность управлять связью имен с методами. В частности, для реализации позднего связывания метод объявляется виртуальным (фактическим) и, таким образом, для данной функции реализуется полиморфизм. Если объявление виртуальности опущено, то метод полностью определяется во время компиляции и не может быть изменен позднее. Способ выбора выполняемого метода подробно описан в тексте, заключенном в рамку.

Наследование и типизация. Рассмотрим переопределение функций send:

```

void ElectricalData :: send () {
    TelemetryData :: send ();
    // transmit the fuelCell1Voltage and the fuelCell2Voltage
    // transmit the fuelCell1Amperes and the fuelCell2Amperes
    // transmit the currentPower
};

```

В большинстве объектно-ориентированных языков программирования допускается переопределение в подклассе метода, который был определен в суперклассе. Из примера видно, что при переопределении метода принято использовать для него также имя, которое было дано в суперклассе. В языках Smalltalk и Object Pascal допускаются ссылки на метод непосредственного предка (суперкласса) с помощью ключевых слов *super* и *inherited* (соответственно для Smalltalk и Object Pascal). В этих же языках определен специальный параметр *self* для указания на объект, связанный с данным методом. В языке C++ для обращения к методу суперкласса (любого суперкласса в пределах «видимости») перед именем метода следует указать имя такого суперкласса. Для указания на сам объект в C++ существует специальный указатель *this*. Для достижения той же цели в языке CLOS определена функция *call-next-method*.

Процесс вызова функции (метода)

В традиционных языках программирования вызов подпрограмм является вполне определенной операцией. Например, в Pascal для вызова подпрограммы P компилятор модифицирует стек, помещая в него определенные аргументы, после чего управление передается на начало кода подпрограммы P. Однако в языках Smalltalk, Object Pascal, C++, CLOS процедура вызова подпрограммы реализуется более сложным динамическим образом, поскольку класс объектов-операндов определяется только во время выполнения программы. Если добавить к этому механизм наследования, то ситуация еще более усложняется. Наследование без полиморфизма мало отличается от обычного вызова подпрограммы, но наличие полиморфизма приводит к более сложным механизмам реализации.

Рассмотрим следующую иерархическую структуру (рис. 3-5), в которой имеется базовый класс Shape и три подкласса с именами Circle, Triangle и Rectangle. Для класса Rectangle определен в свою очередь подкласс Solid Rectangle. Предположим, что в классе Shape определена переменная theCenter (соответствующая координатам X и Y центра изображения в определенной системе координат), а также следующие операции:

- | | |
|--------------|---|
| • Set Center | Установить координаты (X и Y) центра |
| • Draw | Сформировать изображение |
| • Center | Возвращает значение координат (X и Y) центра. |

Операции Set Centre и Center являются общими для всех подклассов и не требуют переопределения. Однако операция Draw для каждого подкласса является индивидуальной и должна быть определена заново (переопределена). Поскольку класс Shape является абстрактным, тело метода Draw является пустым (это чисто виртуальная функция по терминологии C++).

Класс Circle включает переменную the Radius и соответственно операции для установки и чтения значения этой переменной. Для этого класса операция Draw формирует изображение окружности заданного радиуса с центром в определенной точке (X и Y). В классе Rectangle таким же образом введены переменные the Height и the Width и операции установки и чтения их значений. Операция Draw в данном случае формирует изображение прямоугольника заданной высоты и ширины с центром в заданной точке (X и Y). Подкласс Solid Rectangle наследует все особенности класса Rectangle, но операция Draw в этом подклассе переопределена.

В классе Solid Rectangle вначале выполняется операция Draw по методу суперкласса (Rectangle), а затем изображение дополнительно закрашивается. На рис. 3-6 показаны отношения между указанными классами с учетом полиморфизма. Мы видим здесь однородный список изображе-

ний, полагая, что список может содержать объекты подклассов, входящих в класс Shape. Предположим, что некоторый объект-пользователь хочет сформировать всю совокупность изображений из числа находящихся в списке. Будем делать это итеративно, проходя по списку и выполняя метод Draw для каждого объекта. В этом случае компилятор не сможет создать статический код вызова операции Draw, так как класс объектов в списке заранее не определен.

Поскольку язык Smalltalk является нетипизированным, вызов операций будет полностью динамическим. Когда объект-пользователь сформирует сообщение Draw применительно к объекту, встретившемуся в списке, будет происходить следующее:

- Соответствующий объект осуществляет поиск данного сообщения в словаре своего класса.
- Если сообщение найдено, вызывается нужный код для найденного локального метода.
- Если в своем классе сообщение нужного вида не найдено, поиск перемещается в суперкласс.

Этот процесс выполняется для всей иерархии суперклассов, пока не будет найдено нужное сообщение или процесс не достигнет базового класса Object. Если и в последнем случае сообщение не будет найдено, Smalltalk формирует сигнал о наличии ошибки `doesNotUnderstand`.

Основную роль в приведенном алгоритме играет словарь сообщений, являющийся составной частью каждого класса, скрытой от объектов-пользователей.

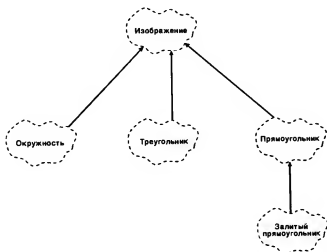


Рис. 3-5. Схема класса Shape.



Рис. 3-б. Схема объекта Shape.

Такой словарь создается вместе с образованием класса и содержит информацию о всех методах, которые могут применяться к объектам-экземплярам данного класса. Поиск в словаре сообщений занимает определенное время, поэтому в языке Smalltalk поиск метода занимает в 1,5 раза больше времени, чем обычный вызов подпрограммы.

Во всех коммерческих версиях языка Smalltalk реализуется оптимизация процесса поиска за счет помещения словаря сообщений в кэш-память. Это улучшает временные характеристики на 20-30% [27]. Операция Draw, определенная в подклассе Solid Rectangle представляет собой особый случай. Мы уже отмечали, что при данной реализации метода Draw вначале делается вызов метода Draw определенного в суперклассе Rectangle. В Smalltalk для вызова метода суперкласса используется ключевое слово Super. Декларирование метода Draw вместе с Super приводит к тому, что алгоритм поиска начинается не с данного класса, а сразу с суперкласса.

Исследования Дейтча дают основание полагать, что время поиска при реализации механизма полиморфизма может быть сокращено на 85%, а передача сообщений при этом сравняется по времени с обычным временем процедур [28]. Даф замечает, что в таких ситуациях программист часто подразумевает реализацию раннего связывания классов объектов [29]. К сожалению, в нетипизированных языках отсутствуют средства, позволяющие сообщить компилятору о предположениях программиста.

В строго типизированных языках типа Object Pascal и C++ такая возможность реализуется. В этих языках алгоритм вызова методов несколько отличается от описанного выше и позволяет сократить, где возможно, время поиска, сохранив при этом свойства полиморфизма.

В C++ операции, подразумевающие позднее связывание, объявляются виртуальными (Virtual), а все остальные обрабатываются компилятором как обычные вызовы подпрограмм. В нашем примере метод Draw должен быть виртуальной функцией, а методы Set Center и Center — обычными

(поскольку они не переопределяются). Невиртуальные методы могут быть объявлены подставляемыми (in Line), при этом соответствующая процедура целиком включается в код программы, создавая эффект макроопределения. Однако макроопределение приводит к дополнительным затратам памяти.

Для управления реализацией виртуальных функций в C++ используется концепция v-таблиц, которые создаются для каждого объекта при его образовании (т.е. для фиксированного класса объекта). Такая таблица содержит список указателей на виртуальные функции. Например, при создании объекта класса Rectangle v-таблица будет содержать графу для виртуальной функции Draw, содержащую указатель на ближайшую по иерархии реализацию функции Draw. Если в классе Shape определена виртуальная функция Rotate, которая в классе Rectangle не переопределена, то соответствующий указатель для Rotate будет связан с классом Shape. В соответствии с этим осуществляется поиск нужной функции во время исполнения программы: происходит косвенное обращение через соответствующий указатель к функции объекта и сразу реализуется правильно выбранный код без всякого поиска [30].

Операция Draw в классе Solid Rectangle представляет собой особый случай в языке C++. Чтобы реализовать метод Draw, определенный в суперклассе, применяется специальный оператор, указывающий на место определения функции. Это выглядит следующим образом:

```
Rectangle :: Draw();
```

Исследование Страустрапа показали, что вызов виртуальной функции по эффективности немного уступает вызову обычной функции [31]. Для простого наследования вызов виртуальной функции требует дополнительно выполнения трех-четырех операций адресации по отношению к обычному вызову; при множественном наследовании число таких дополнительных операций адресации составляет пять или шесть.

Существенно сложнее выполняется поиск нужных функций в языке CLOS, здесь используются дополнительные квалификаторы: :before, :after, :around. Операции определяются как обобщенные функции и каждая такая функция может быть связана с множеством методов. Наличие множественного полиморфизма еще более усложняет проблему. При выборе нужного метода в языке CLOS, как правило, реализуется следующий алгоритм:

- * Определяется тип аргументов.
- * Устанавливается множество допустимых методов.
- * Методы сортируются в направлении от наиболее специализированных к более общим и в соответствии со списком старшинства классов.
- * Выполняются вызовы всех методов с квалификатором :before.
- * Выполняется вызов наиболее специализированного первичного метода.
- * Выполняются вызовы всех методов с квалификаторами :after.
- * Возвращается значение первичного метода [32].

Язык CLOS реализует особый прием программирования метаобъектов, который позволяет переопределять любые алгоритмы, используемые для управления обобщенными функциями. На практике, однако, используют в основном системные алгоритмы в исходном виде. Как справедливо отметили Уинстон и Хорн: «Алгоритмы, используемые в языке CLOS, сложны, и даже кудесники программирования стараются не вникать в их особенности, так же как физики предпочитают иметь дело с механикой Ньютона, а не квантовой механикой» [33].

На практике часто приходится обращаться к методам суперклассов. Подклассы играют роль расширителя возможностей суперклассов. В языке CLOS явно определены квалификаторы *before*, *after*, *around*, определяющие разновидность метода. Метод без квалификатора считается исходным (главным) и определяет в основном поведение объекта. Методы с квалификаторами *before* и *after* вызываются соответственно до и после главного метода с помощью функции *call-next-method*.

Все подклассы на рис. 3-4 являются подтипами исходного класса, т.е. экземпляры классов *ElectricalData* и *TelemetryData* являются подтипами. Для всех строго типизированных языков, включая Object Pascal и C++, является характерным наличие параллелизма в отношениях типизации и наследования. К языкам Smalltalk и CLOS это относится в гораздо меньшей степени из-за отсутствия в них типизации.

Параллель между типизацией и наследованием следует ожидать при создании иерархии по принципу обобщения/специализации, где механизм наследования позволяет реализовать смысловые связи между абстракциями. Рассмотрим следующее описание на C++:

```
TelemetryData telemetry;
ElectricalData electrical (5.0, -5.0, 3.0, 7.0);
```

Следующий оператор присвоения является правильным:

```
telemetry = electrical; //electrical is a subtype of telemetry
```

Но следующий оператор является ошибкой:

```
electrical = telemetry; // illegal: telemetry is not a subtype of electrical
```

Можно сделать заключение, что присвоение объекту *Y* значения объекта *X* допустимо, если тип объекта *X* совпадает с типом объекта *Y* или является его подклассом.

В большинстве строго типизированных языков программирования допускается преобразование значений из одного типа в другой, но только в тех случаях, когда между двумя типами существуют отношения вида класс/подкласс. Например, в языке C++ допускается введение операторов явного преобразования типов (классов), называемого приведением типов (*type cast*). В языке Object Pascal аналогичный механизм носит название «привязка типов» (*type coercion*). Как правило, такие преобразования используются по отношению к объекту специализированного класса, чтобы присвоить его значение объекту более общего класса. Приведение типов не нарушает принцип типизации, поскольку во время компиляции осуществляется необходимый семан-

тический контроль. Иногда необходимы операции приведения объектов более общего класса к специализированным классам. Эти операции не являются надежными с точки зрения строгой типизации, так как во время выполнения программы может возникнуть несоответствие (несовместимость) приводимого объекта с новым типом. Однако такие преобразования достаточно часто используются в тех случаях, когда программист хорошо представляет себе все типы объектов. Например, в случае невозможности определения параметризованного типа очень часто создаются классы, объединяющие множества из различных объектов. Чтобы реализовать возможность произвольного объединения различных классов в такие множества, вводится понятие единого исходного (базового) класса, такого, как `TObject` в языке `Object Pascal`. Для итеративных операций, определенных в базовом классе, существенным является только способ возврата значения объекта этого класса. На практике, однако, допускается объединение в общее множество только объектов определенных подклассов базового класса `TObject`. Чтобы выполнить по отношению к объекту специфические операции итеративно, необходимо явно привязать объект к ожидаемому типу. При этом также существует опасность ошибки исполнения программы, если среди множества объединенных объектов окажется объект недопустимого типа.

В строго типизированных языках, в отличие от нестрого типизированных, методы разрешения неопределенности (поиска функции) лучше оптимизировать. В этом случае передача сообщений занимает не больше времени, чем вызов обычной подпрограммы. Однако в обеспечении параллельности иерархии типов и иерархии наследования есть слабые места. В частности, модификация структуры и поведения какого-либо суперкласса может повлиять на структуру подклассов. Мнкаллеф утверждает: «Если правила типизации основаны на наследовании, то изменения реализации класса, затрагивающие его место в иерархической структуре, могут нарушить соответствие типов класса-пользователя даже при неизменном строении интерфейса класса» [35]. Это ставит под вопрос цель механизма наследования. Мы уже говорили выше, что цель наследования — разделение ресурсов или установление смысловых связей между объектами. Другого подхода придерживается Снайдер: «Можно рассматривать наследование как особый способ макроопределения кода, что может быть полезным; такой способ обеспечивает простоту внесения изменений. С другой стороны, на наследование можно смотреть как на явное указание логической (смысловой) связи порожденного класса с породившим его классом. При этом порожденный класс является лишь более специализированным или частично переопределенным» [36]. В языках `Smalltalk`, `Object Pascal` и `CLOS` объединены оба этих подхода. В языке `C++` программист имеет более сильные средства использования наследования. В частности, если суперкласс данного подкласса определен в качестве общедоступного (`public`), как в примере с классами `ElectricalData`, то это означает, что подкласс одновременно является подтипом этого суперкласса (поскольку интерфейс является общим, то структура и поведение являются также общими).

Если же суперкласс объявлен обособленным (`private`) в описании подкласса, то подкласс не будет подтипом суперкласса, хотя структура и поведение будут общими. Следовательно, в случае обособленного суперкласса вся структура суперкласса становится в подклассе обособленной (`private`). В этом случае два класса (суперкласс и подкласс) не обладают по отношению к другим классам одинаковым интерфейсом, что означает отсутствие отношения тип-подтип.

Дадим следующее определение класса:

```
class InternalElectricalData : private ElectricalData {
public:
    InternalElectricalData (float v1, float v2, float a1, float a2);
    InternalElectricalData (const InternalElectricalData&);
    virtual ~InternalElectricalData ();
    ElectricalData :: currentPower;
};
```

В приведенном описании суперкласс `ElectricalData` объявлен обособленным, следовательно, методы определенного класса `Internal ElectricalData`, например `Send`, являются закрытыми для всех пользователей. Поскольку класс `InternalElectricalData` не является подтипом для `ElectricalData`, мы уже не сможем присвоить значения объектов этих классов, как в случае декларирования суперкласса в качестве общедоступного. Отметим, что функция `currentPower` сделана видимой для всех объектов-пользователей путем явного указания ее наименования. В другом случае она осталась бы обособленной. Очевидно, что в языке C++ невозможно сделать какой-либо элемент подкласса более «видимым», чем такой же элемент суперкласса. Так, элемент `id`, объявленный в классе `TelemetryData` как защищенный, не может быть сделан в подклассе общедоступным путем явного наименования (как в случае `currentPower`).

В языке Ada для достижения аналогичного эффекта вместо подтипов используется механизм производных типов. Определение подтипа не означает появление нового типа, а лишь определенное ограничение существующего. Определение производного типа создает самостоятельный новый тип, который имеет структуру, заимствованную у исходного типа. В следующем разделе показано, что применение отношений наследования в качестве макроопределений и отношений использования вызывает серьезные противоречия.

Понятие множественного наследования между классами. Мы рассмотрели вопросы, связанные с простым наследованием, когда подклассы имеют только один исходный суперкласс. Однако, как указали Влнсайдес и Линтон: «простое наследование при всей своей полезности часто заставляет программиста выбирать между двумя равнопривлекательными классами. Это ограничивает возможность повторного использования переопределенных классов и заставляет дублировать уже имеющиеся коды. Например, затруднительно в этой ситуации собрать воедино изображение окружности и какой-либо рисунок; необходимо выбрать что-то одно и дописать заново структуру класса, не вошедшего в наследованный класс» [37]. Множественное наследование реализуется в языках C++ и CLOS, а также частично в Smalltalk. Необходимость реализации множественного наследования в ООП остается предметом горячих споров. Практика говорит о том, что множественное наследование играет роль парашюта: в нем нет постоянной необходимости, но если он вдруг понадобился, то большое счастье иметь его под рукой.

Предположим, что возникла необходимость классифицировать различные продукты: бананы, овсяные хлопья, тесто, молоко и говядину. Можно распределить их по четырем основным группам: овощи и фрукты, мучные продукты, молочные продукты, мясо. Далее отношения устанавливаются путем наследования в виде иерархии «по номенклатуре»: молоко — разновидность молочных продуктов, которые в свою очередь являются видом пищевых продуктов.

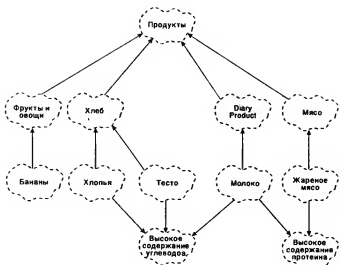


Рис. 3-7. Отношения множественного наследования между классами.

Но есть другие способы классификации продуктов. Например, можно классифицировать их по содержанию белков и углеводов. Для такой классификации простого наследования оказывается недостаточным, и мы приходим к понятию множественного наследования.

На рис. 3-7 показана такая структура классов. Здесь мы видим, что овсяные хлопья являются видом мучных продуктов и одновременно источником углеводов. Говядина является разновидностью мяса и источником белков. Молоко находится в этой классификации в особом положении: оно относится к числу молочных продуктов и является источником и углеводов, и белков. Реализация указанного подхода на языке CLOS состоит в определении базового класса Food и четырех абстрактных классов, соответствующих группам продуктов:

```

(defclass food () (...))
(defclass fruit-and-vegetable (food) (...))
(defclass bread (food) (...))
(defclass dairy-product (food) (...))
(defclass meat (food) (...))

```

Содержание слотов в определении каждого из классов для краткости опущено. Затем определяются классы, соответствующие продуктам питания:

```

(defclass high-carbohydrate () (...))
(defclass high-protein () (...))

```

Наконец, вводятся классы разновидностей продуктов:

```
(defclass banana (fruit-and-vegetable) (...))
(defclass bran-flakes (break high-carbohydrate) (...))
(defclass pasta (break high-carbohydrate) (...))
(defclass milk (dairy-product high-carbohydrate high-protein) (...))
(defclass beef (meat high-protein) (...))
```

Каждый из этих классов имеет несколько исходных суперклассов. Разработка структуры классов на основе наследования (особенно множественного) — задача достаточно трудная. В гл. 2 мы говорили, что этот процесс обычно является ступенчатым и итеративным. При множественном наследовании появляются две дополнительные проблемы: как быть в случае неопределенности в наименовании элементов различных суперклассов и как реализовать повторное наследование.

Неопределенность наименования имеет место в тех случаях, когда в нескольких суперклассах используются одинаковые имена для обозначения элементов интерфейса (переменных или методов). Предположим, например, что в двух классах *high-carbohydrate* и *high-protein* имеется слот с именем *percentage*, означающий процент соответственно углеводов и белков в продукте. Поскольку класс *milk* наследует структуру этих двух классов, то возникает вопрос: а как быть с двумя слотами, имеющими одинаковое имя? Эта проблема может быть разрешена тремя основными путями. Во-первых, можно запретить при компиляции такой тип наследования. Такой подход реализован в языках *Smalltalk* и *Eiffel*. Однако в языке *Eiffel* можно переименовать один из слотов и избежать данной неопределенности. Во-вторых, семантика языка может допускать наличие двух одинаковых систем в разных классах, рассматривая их как один слот; такой подход реализован в языке *CLOS*. В-третьих, можно допускать смешение имен при наличии дополнительного квалификатора, указывающего на базовый суперкласс для данного имени; так сделано в *C++*. Вторая проблема связана с повторным наследованием и характеризуется Мейером так: «одним из таких мест в связи с множественным наследованием является проблема неоднократного использования одного предка (исходного суперкласса) в каком-либо классе. Если множественное наследование допустимо, то рано или поздно кем-то будет объявлен класс *D*, имеющий два суперкласса *B* и *C*, которые в свою очередь построены на базе суперкласса *A* — или каким-либо другим способом, когда *D* дважды (или более) наследует от суперкласса *A*. Эта ситуация называется повторным наследованием и требует должного внимания» [38]. Рассмотрим следующий пример:

```
(defclass breakfast-cereal (bran-flakes milk) (...))
```

Данный класс повторно наследует класс *high-carbohydrate*, являющийся суперклассом для *bran-flakes* и *milk*.

Проблема повторного наследования решается тремя способами. Во-первых, можно запретить наличие повторного наследования. Так сделано в языках *Smalltalk* и *Eiffel* (но в *Eiffel* допускается переименование для устранения неопределенности). Во-вторых, можно потребовать указания дополни

тельного квалификатора, определяющего порядок наследования, как сделано в C++. В-третьих, можно трактовать многократное указание на некоторый класс как обозначение данного класса. В языке C++ повторное введение суперкласса соответствует определению виртуального базового класса. Виртуальный базовый класс возникает тогда, когда в некотором подклассе вводится имя суперкласса с указанием признака виртуальности, что указывает на использование ресурсов такого класса. В языке CLOS также используются повторные классы на основе списка следования классов. Этот список дополняется при введении каждого нового определения класса и учитывает все суперклассы для каждого класса, исключая повторения и реализуя следующие правила:

- * Любой класс имеет предшественников в лице его суперклассов.
- * Для каждого класса устанавливается порядок следования его непосредственных суперклассов [39].

При этом подходе схема наследования выравнивается, повторения устраняются и в результате получается структура, соответствующая простому наследованию [40]. Этот процесс можно трактовать как топологическую сортировку классов. Если общее упорядочение структуры классов реализуется, то наличие классов с повторным наследованием допустимо. Следует отметить, что процесс упорядочения может быть реализован либо единственным образом, либо иметь несколько вариантов. Если найти такой порядок не удастся (например, когда имеются перекрестные и циклические зависимости между классами), то описание класса отвергается как ошибочное. В приведенном примере класс *breakfast-cereal* приемлем, поскольку следование суперклассов однозначно упорядочено; в иерархию суперклассов класс *high-carbohydrate* входит только один раз.

Множественное наследование привело к возникновению такой разновидности классов как примеси (Mixins). Этот термин связан с традициями программирования на языке Flavors: здесь допускается объединять (смешивать) более мелкие классы в более сложные. Хендлер утверждает: «Смешение напоминает по синтаксису регулярный класс, но имеет совершенно другую цель. Цель этого вида классов состоит единственно в ... добавлении функций по отношению к другим объектам [классам] — экземпляры классов-смесей никогда не создаются» [41]. Классы *high-carbohydrate* и *high-protein* являются смешиваемыми. Эти классы не нужны сами по себе, а используются для наполнения других классов особыми свойствами. В языке CLOS принято осуществлять смешение за счет квалификаторов: *before* и *after*, чтобы дополнить исходный метод иужными уточнениями. Таким образом, можно определить примесь как некоторый класс, реализующий особый вид поведения, и использовать этот класс для внесения его поведения в другие классы (через механизм наследования). Как правило, поведение смешиваемого класса противоположно поведению классов, с которыми осуществляется соединение.

Класс, образованный исключительно путем наследования примесей без всяких добавлений, называется *агрегированным* классом.

Понятие множественного полиморфизма. Определим следующую обобщенную функцию:

```
(defgenetic display (food))
```

8 Гради Буч

Эта функция предназначена для создания изображения продукта в графическом окне. В каждом подклассе должен быть описан метод, соответствующий этой обобщенной функции. Таким образом, начинает реализовываться механизм полиморфизма — если мы будем реализовывать указанный метод по отношению к конкретному объекту, возникнет соответствующее изображение (сам объект определяет какому классу он принадлежит). Такой полиморфизм является простым, поскольку выбор нужного метода осуществляется на основе одного параметра.

Допустим теперь, что нам нужно корректировать поведение метода в соответствии с типом используемого дисплейного устройства. В одном случае мы будем рисовать изображение с помощью метода `display`, а в другом печатать соответствующее текстовое сообщение. Для этого нам придется создать две различные, но очень близкие, обобщенные функции. Такое решение нельзя считать удовлетворительным, так как описание избыточно.

Язык CLOS позволяет определить методы, специализированные по нескольким параметрам; такие методы называются *множественными* (*multi-methods*). Определим таким образом следующую функцию:

```
(genetic display (food display-device))
```

Прежде чем вызывать такую функцию, нужно создать экземпляр одного из подклассов `food` и экземпляр подкласса `display-device`. Язык CLOS позволяет при вызове такой функции выбрать нужный исходный метод, соответствующий действительным параметрам. Если найти такой метод не удастся, то возникает сообщение об ошибке исполнения. Так реализуется множественный полиморфизм.

Отношения использования

Пример отношений использования между классами. Не всегда с помощью механизма наследования удастся адекватно отразить всю совокупность сложных отношений между абстракциями. Рассмотрим, например, отношения между библиотекой и книгами. Библиотека не является разновидностью книги, а объединяет их. На языке *Object Pascal* можно дать следующее определение:

```
TLibrary = object (TObject)
...
  procedure TLibrary.Initialize;
  procedure TLibrary.Checkout (ABook : TBook);
  procedure TLibrary.Checkin (ABook : TBook);
...
end;
```

Часть методов и полей данного класса для краткости опущена. Компиляцию описания этого класса можно осуществить только после компиляции класса `TBook`, поскольку он входит в интерфейс класса `TLibrary`. Для класса `TLibrary` использование класса `TBook` означает, что `TLibrary` «видим» для `TBook`, а интерфейс и реализация `TLibrary` могут обращаться к интерфейсу (но не к реализации) `TBook`. Например, реализация метода `Checkout` подразумевает посылку сообщения объекту `ABook`, чтобы нужным образом изменить состояние этого объекта (установить метку о прохождении контроля). Таким образом, можно видеть, что отношения использования между классами

ми похожи, но отличаются от отношений использования между объектами. Объекты класса TLibrary (и его подклассов) могут посылать сообщения объектам класса TBook (и его подклассам).

Понятие отношений использования между классами. Отношения использования имеют два различных аспекта: в интерфейсной части одного класса может быть использован другой класс (как в предыдущем примере) или другой класс используется в реализации. В первом случае используемый класс должен находиться в зоне «видимости» классов пользователей. Например, код функции, выполняющей контроль книг в библиотеке, должен быть «видимым» как для класса TLibrary (здесь происходит вызов функции Checkout), так и для TBook (чтобы сослаться на объект этого класса).

Во втором случае используемый класс находится в области ограниченного доступа использующего класса. Например, в реализации класса TLibrary может быть употреблен класс TList, представляющий собой список книг. Для класса TList необязательно соблюдать требование «видимости» для интерфейса TLibrary, достаточно выполнить условие «видимости» для реализации TLibrary. Рассмотрим ситуацию, когда любая библиотека может состоять из n книг, но каждая книга имеется только в одной из библиотек. Это пример отношения вида 1:п. В другом случае можно допустить, что в каждой библиотеке имеется определенный набор (коллекция) книг и каждая коллекция существует только в одной библиотеке. Это пример отношения вида 1:1. Возможны и числовые отношения других видов, например $m:n$.

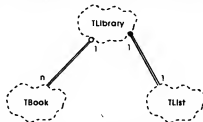


Рис. 3-8. Отношения использования между классами.

Два основных вида отношений использования показаны на рис. 3-8. Обратим внимание на способ отражения разновидностей отношения (использование класса в интерфейсной части описания или в реализации) и числовых отношений.

В гл. 2 было показано значение ограничения доступа к данным. Однако возможны случаи, когда ограничение доступа становится препятствием, особенно для отношений использования классов. Допустим, что мы создали класс TSortedBookList, который используется по отношению к объектам TBook для осуществления эффективной сортировки. В большинстве языков ООР существуют строгие правила защиты информации. В языке C++ реализована возможность ослабить действие таких ограничений путем объявления общности (дружественности) классов. Объявление общности (friend) позволяет использовать метод двумя и более объектами разных классов. При этом реализация метода для одного из классов может находиться в обособленной части класса, который также является дружественным. Другими словами, установление отношений общности для структуры и методов класса означает

возможность доступа к тем элементам класса, которые в другом случае являются обособленными. Как и в обычной жизни, к выбору друга нужно подходить осторожно, поскольку такие отношения подразумевает доверие, которое применительно к классам означает гарантию неприкосновенности данных, оставшихся незащищенными.

Множественное наследование часто приводит к проблемам в отношениях использования. Можно, например, определить класс *telephone* путем наследования классов, соответствующих клавиатуре, микрофону и громкоговорителю. Можно, наоборот, определить этот же класс путем использования трех указанных классов. В обоих случаях достигается различными путями один и тот же эффект. Опыт показывает, что использование множественного наследования для агрегатирования (смысл которого в том, что один объект включает несколько других) не эффективно. Телефон не является разновидностью микрофона, а содержит его в себе (использует). Поэтому использование объектов больше подходит для агрегатирования. Общее правило гласит: если некоторая абстракция представляет нечто большее, чем сумму некоторых абстракций — пригодны отношения; если абстракция является подвидом другой абстракции или соответствует простой сумме компонент, следует использовать отношение наследования.

Отношение наполнения

Примеры отношений наполнения между классами. В строго типизированных языках принято использовать соответствующие особенности проектируемой системы — типы данных. Предположим, что мы умело пользуемся абстракцией универсала с десятью контрольными классами. Для представления контрольных классов в языках *Object Pascal* и *Ada* можно применить специфический ограниченный тип целых чисел с диапазоном от 1 до 10. На более высоком уровне абстракции нужно образовать множественный класс для обозначения группы служащих, произвольно располагаемых по контрольным классам. Здравый смысл подсказывает, что в эту группу будут входить только служащие универсала, но не покупатели и уж совсем не овощи. По логике нам следует точно определить этот множественный класс (т.е. класс входящих в него объектов) и этим гарантировать от попадания в него других объектов (средствами языка программирования).

Множество является примером сборного класса, т.е. класса, экземпляры которого состоят из наборов других объектов. Сборные классы могут быть однородными (состоять из объектов одного класса) или неоднородными (состоять из объектов разных классов, имеющих общий суперкласс). Наиболее часто встречаются такие виды сборных классов, как стек, список, строка, очередь, двухсторонняя очередь, замкнутая цепь, маршрут (план), множество, выборка, дерево и граф [42].

На языке *Ada* множественный класс определяется следующим образом:

```
generic
  type Item is private;
package Simple_Set is
  type Set is limited private;
  procedure Copy      (From_The_Set : in      Set;
                       To_The_Set   : in out Set);
  procedure Clear     (The_Set      : in      Set);
```

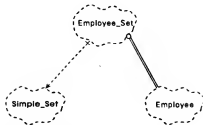


Рис. 3-9. Отношения наполнения между классами.

```

procedure Add          (The_Item      : in      Item;
                        To_The_Set    : in      Set;
procedure Remove      (The_Item      : in      Item;
                        From_The_Set  : in      Set;
procedure Union        (Of_The_Set    : in      Set;
                        And_The_Set   : in      Set;
procedure Intersection (Of_The_Set    : in      Set;
                        And_The_Set   : in      Set;
                        To_The_Set    : in      Set;
procedure Difference   (Of_The_Set    : in      Set;
                        And_The_Set   : in      Set;
                        To_The_Set    : in      Set;

function Is_Equal      (Left           : in Set;
                        Right          : in Set) return Boolean;
function Expent_Of     (The_Set        : in Set) return Natural;
function Is_Empty      (The_Set        : in Set) return Boolean;
function Is_A_Member   (The_Item      : in Item;
                        Of_The_Set     : in Set) return Boolean;
function Is_A_Subset   (Left           : in Set;
                        Right          : in Set) return Boolean;
function Is_A_Proper_Subset (Left       : in Set;
                              Right     : in Set) return Boolean;

Overflow          : exception;
Item_Is_In_Set    : exception;
Item_Is_Not_In_Set : exception;

```

```
private
```

```
...
end Simple_Set;
```

Вновь обратим внимание на стиль описания интерфейсной части класса, где явно выделены модификаторы и селекторы. Используя этот класс и класс Employee, можно осуществить наполнение класса объектами, которые принадлежат классу Employee:

```
packade Employee_Set is new Simple_Set (Item => Employee);
```

Полагая, что имя множественного объекта Available-Employees, а имена служащих, входящих в это множество, Mike, Paul, Dave, Bob и Brett, запишем следующие выражения:

```
Add (Bob, To_The_Set => Available_Employees);
Add (Paul, To_The_Set => Available_Employees);
```

```
Add (Brett, To_The_Set => Available_Employees);
Remove (Mike, From_The_Set => Available_Employees);
if Is_A_Member (Dave, Of_The_Set => Available_Employees then ...
```

Правила строгой типизации языка Ada позволяют отвергнуть все попытки включить в это множество и выполнить другие операции с объектами другого класса (кроме класса, представляющего служащих).

Значение отношений наполнения между классами. Существует четыре основных способа построения сборного класса. Первый способ — использование макроопределений. Он применяется в C++, но по мнению Страустрапа, «этот подход хорош только в небольших проектах» [43], поскольку использование макросов неудобно; более того, каждая реализация при этом создает новый вариант (версию) кода. Второй способ реализован в языке Smalltalk и включает исследование и позднее связывание [44]. По этому методу могут создаваться только однородные сборные классы, поскольку каждый элемент рассматривается как экземпляр базового класса Object. Третий способ является традиционным для языка Object Pascal. Здесь, как и в Smalltalk, создается обобщенный сборный класс, но затем используется специальная процедура контроля типа, которая позволяет в процессе образования объекта закрепить за ним определенный класс элементов. Четвертый способ заключается в механизме параметризованного класса, впервые реализованном в языке CLU [45]. Параметризованный класс (называемый также обобщенным классом) — это класс, составляющий основание для размещения других классов. Он параметризуется классами, объектами и операциями. Параметризованный класс необходимо наполнить (конкретизировать параметры этого класса) прежде, чем создавать объекты. Языки Ada и Eiffel реализуют механизм параметризации, а в C++ ожидается появление этого средства в ближайшем будущем.

Отношения наполнения классов иллюстрируются рис. 3-9. Отметим, что для наполнения класса Simple-Set используется класс Employee. Отношения наполнения почти всегда сопровождаются отношениями использования, при этом происходит выявление действительных классов, наполняющих класс-основание.

Мейер доказал, что механизм наследования является более мощным средством, чем механизм конкретизации/обобщения, и многие свойства обобщения достигаются путем наследования, но не наоборот [46]. На практике всегда полезно иметь дело с языками, реализующими оба механизма — наследования и параметризации.

Параметризованные классы используются не только для построения сборных классов. Страустрап утверждает, что «параметризация типов позволит реализовать арифметические функции на основе базового параметра, а в конечном счете создать унифицированные функции для любых типов параметров: целых, действительных, двойной точности и т.д.» [47]. С точки зрения поддержки системного проектирования параметризованные классы полезны для описания интерфейсов. При описании операций над объектами можно реализовать подобие шаблона, позволяющего реализовать определенные действия над объектами разных классов. Примером может служить абстракция упорядоченного списка, объекты которого должны сортироваться по некоторому критерию. Метод, служащий для определения положения объекта в списке, может быть параметризован. При этом создается операция, которая будет выполняться над элементами списка, представляющими объекты разных классов. Параметризация класса делает его более свободным (универ-

сальным) и, следовательно, пригодным для более широкого использования. Класс становится менее специализированным, может конкретизироваться объектами любых других классов.

Отношения типа метакласс

Значение отношений типа метакласс. Три рассмотренных выше типа отношений между классами — наследование, использование и наполнение — обеспечивают практически все потребности программистов при описании взаимоотношений классов. Однако из идеологии объектного подхода вытекает еще одна разновидность отношения абстракций. Мы уже говорили, что каждый объект является экземпляром определенного класса (реализацией класса). А если попытаться сделать объектом манипулирующий сам класс? Для ответа на этот вопрос нужно определить, что такое класс классов. Очевидно, что это метакласс. Другими словами, метакласс — это класс, экземпляры которого сами являются классами.

Мотивируя необходимость метаклассов Робсон отмечает, что «при разработке системы взаимодействие объектов обеспечивается через интерфейс классов. По этой причине чрезвычайно полезно обеспечить возможность манипулирования классами так же, как любыми другими объектами» [48]. Метаклассы в непосредственном виде реализуются в языках CLOS и Smalltalk, причем в языке CLOS этот механизм особенно сильно развит.

В языке Smalltalk метаклассы используются главным образом для инициализации переменных класса и создания одиночных экземпляров метаклассов [49]. Язык Smalltalk традиционно снабжается рядом примеров, демонстрирующих способы применения классов на основе метаклассов.

Примеры использования метаклассов. Предположим, что определен класс Timer, объекты которого содержат данные о текущем времени, отсчитываемом в секундах от момента активизации системы. Допускается создание произвольного числа таких объектов, но каждый из них должен обеспечивать отсчет одного и того же значения времени. Это означает наличие общего элемента в описании состояния этих объектов и необходимость предварительной инициализации данного элемента. Как правило, состояние объекта определяется переменными объекта, но можно использовать для этого и переменные класса. Переменные класса имеют такой же смысл, как переменные объекта, за исключением того, что их значение является общим для всех экземпляров данного класса. Теперь определим на языке Smalltalk класс Timer и соответствующий ему метод:

```
Object subclass: #Timer
  InstanceVariableNames: ''
  classVariableNames: 'ElapsedSeconds TimesProcess'
  poolDictionaries: ''
  category: 'Simulation'
```

Times methodsFor: 'accessing'

elapsedSeconds

«Return an Integer value representing the number of seconds that have elapsed since the system was activated.»
^ElapsedSeconds

В качестве переменных класса декларированы ElapsedSeconds и TimerProcess, следовательно, они будут общими для всех объектов класса Timer. Для осуществления инициализации этих двух переменных необходимо создать метакласс Timer:

```

Timer class
  instanceVariableNames: ''

Timer class methodsFor: 'initialize-release'
initialize
  «Reset elapsedSeconds, then start the process to update elapsedSeconds every second.»
  ElapsedSeconds <— 0.
  TimesProcess <— [(Delay forSeconds: 1) wait.
                    self elapsedSeconds: self elapsedSeconds + 1.
                    true]
                    whileTrue] newProcess.
  TimesProcess resume

release
  «Stop the process to update elapsedSeconds every second.»
  TimesProcess terminate

```

Метод initialize называется *методом класса*. Осуществление этого метода заключается в создании нового процесса (Timer Process), который через каждую секунду увеличивает значение переменной elapsedSeconds.

В языке Smalltalk каждому классу может соответствовать только один метакласс (метакласс для класса Timer назван Timer class), но не для всех классов в процессе проектирования создаются метаклассы (это определяется спецификой предметной области). В приведенном примере прежде всего необходимо инициализировать класс Timer с помощью следующего оператора:

```
Timer initialize.
```

Поскольку Timer является классом, то процедура initialize должна быть определена в его классе, т.е. в метаклассе Timer class. Этот механизм иллюстрируется рис. 3-10.

Что такое класс для метакласса? В языке Smalltalk любой метакласс является реализацией класса Metaclass (в том числе и Metaclass class). Metaclass является подклассом ClassDescription, который в свою очередь является подклассом Behavior, а последний, наконец, подклассом базового класса Object.

Язык C++ в явном виде не реализует метаклассы, но позволяет создавать переменные класса и методы класса. В частности, можно использовать квалификатор static с переменными и функциями класса, чтобы сделать их общими для всех экземпляров данного класса.

Как уже говорилось, в языке CLOS механизм образования метаклассов наиболее развит. Здесь метаклассы позволяют переопределять семантику любых элементов, таких, как старшинство классов, обобщенные функции и методы. Основным преимуществом такого мощного механизма является возможность экспериментировать с новыми парадигмами GOP и создавать инструментальные средства программирования с многооконным интерфейсом.

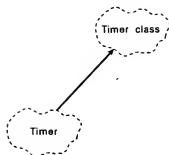


Рис. 3-10. Отношения типа метакласс.

В языке CLOS предопределен класс `standard-class`, являющийся метаклассом для всех классов, определенных с помощью квалификатора `defclass`. В этом общем метаклассе определен метод `make-instance`, определяющий семантику образования экземпляров. Здесь же (в `Standard-class`) определен алгоритм ведения списка старшинства классов. Оба указанных алгоритма в языке CLOS можно переопределить.

В языке CLOS будем рассматривать в качестве объектов метода и обобщенные функции. Поскольку это будут не совсем обычные объекты, введем специальный термин «метаобъекты», включающий в себя объекты классы, объекты методы и объекты обобщенные функции. Всякий метод является реализацией предопределенного класса `standard-method`, а обобщенная функция — реализацией класса `standard-generic-function`. В языке CLOS допускается переопределение всех методов в предопределенных классах, а следовательно, можно изменить поведение всех методов и обобщенных функций.

3.5. ВЗАИМОСВЯЗЬ КЛАССОВ И ОБЪЕКТОВ

Отношения между классами и объектами

Классы и объекты — тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса, а класс может порождать любое число объектов. В большинстве практических случаев классы статичны, т.е. все их особенности и содержание определены в процессе компиляции программы. Из этого следует, что любой созданный объект относится к строго фиксированному классу. Объекты, наоборот, в процессе выполнения программы непрерывно создаются и разрушаются.

В качестве примера рассмотрим классы и объекты для задачи управления воздушным движением. Наиболее важные абстракции в этой задаче — самолеты, графики полетов, маршруты, воздушное пространство и его распределение. Эти классы объектов по смыслу достаточно статичны. Такая статичность необходима, иначе невозможно решить задачу перелета из одного места в другое так, чтобы два самолета не оказались одновременно в одной точке пространства.

Объекты этих классов, наоборот, весьма изменчивы и динамичны. Новые маршруты полетов возникают не так часто. Существенно быстрее изменяются типы самолетов, находившихся в эксплуатации. Быстрота, с которой самолеты занимают и покидают воздушные коридоры, имеет еще большую динамику.

Роль классов и объектов в процессе проектирования

На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

- Выявление классов и объектов, составляющих словарь предметной области.
- Построение структур, обеспечивающих совместное взаимодействие объектов, при котором достигаются заданные требования.

В первом случае говорят о ключевых абстракциях задачи (совокупность классов и объектов), во втором — о механизмах реализации (совокупность структур). В первой фазе проекта внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический каркас системы, состоящий из структуры классов и структуры объектов. На последующих фазах проекта, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и механизмов и охватывает вопросы их физического представления. Принимаемые в процессе проектирования решения составляют архитектуру модулей системы и архитектуру процессов в системе.

3.6. ВОПРОСЫ КАЧЕСТВА ПРИ СОЗДАНИИ КЛАССОВ И ОБЪЕКТОВ

Определение качества абстракций

По мнению Ингалса: «Для построения системы должен использоваться минимальный набор неизменяемых компонент; сами компоненты должны быть по возможности стандартизованы и связаны единым способом построения» [50]. Применительно к OOD такими компонентами являются классы и объекты, отражающие ключевые абстракции системы, а единство построения (каркас) обеспечивается соответствующими механизмами реализации.

Опыт показывает, что процесс выделения классов и объектов является последовательно-итеративным. В действительности, за исключением самых простых задач, с первого раза не удастся окончательно выделить и описать классы. В гл. 4 и 7 показано, как в процессе работы происходит сглаживание противоречий, возникающих при начальном определении абстракций. Очевидно, такой процесс связан с дополнительными затратами на перекомпиляцию, согласование и внесение изменений в проект системы. Очень важно, следовательно, с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев:

- Взаимозависимость
- Связность
- Достаточность
- Полнота
- Простота (безызыточность)

Термин «взаимозависимость» (coupling) заимствован из структурного проектирования, но в более вольном толковании он используется и в OOD. Стивейс, Майерс и Константин определяют его так: «степень глубины связей между отдельными модулями. Фрагменты системы, сильно зависящие от других, гораздо сложнее воспринимать, заменять и модифицировать. Для улучшения качества системы следует по возможности избегать сильной зависимости между отдельными модулями» [51]. Пример правильного подхода к проблеме взаимозависимости приведен Пэйдж-Джонсом в виде модульной стереосистемы, где усилитель мощности размещен в конструкции колонки громкоговорителей [52].

Кроме взаимозависимости модулей в OOD существенным фактором является зависимость между классами и объектами. Существует определенное противоречие между явлениями зависимости и наследования. С одной стороны, желательно избегать сильной взаимозависимости классов; с другой стороны, механизм наследования — тесно связывающий подклассы с суперклассами — помогает выгодно использовать сходство абстракций.

Понятие связности также заимствовано из структурного проектирования. Связность — это степень взаимодействия между элементами отдельного модуля (а для OOD еще и отдельного класса или объекта). Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Для примера можно вообразить класс, соединяющий абстракции собак и космических аппаратов. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели. Так, например, класс «собака» будет функционально связанным, если он описывает поведение собаки, собаки в целом и ничего, кроме собаки.

К идеям взаимозависимости и связности тесно примыкают понятия достаточности, полноты и простоты. Под *достаточностью* подразумевается наличие в классе или модуле программы всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Для примера рассмотрим класс «множество». Операция удаления элемента из множества в этом классе, очевидно, необходима, но будет ошибкой не включить в этот класс и операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается класс-пользователь такой абстракции. Под *полнотой* подразумевается наличие в интерфейсной части класса всех необходимых характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все существенные аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все необходимое для взаимодействия с пользователями. Полнота является субъективным фактором, и разработчики часто ей злоупотребляют, вынося на верхний уровень такие операции, которые можно реализовать на более низком уровне. Из этого вытекает требование простоты (минимальной необходимости). *Простоты* являются только такие операции, которые обеспечивают реализацию эффективного действия абстракции. Так, в примере с «множеством» операция добавления элемента является примитивной, а операция добавления четырех элементов не будет примитивной, так как эффективно реализуется через операцию добавление одного элемента. Конечно, эффективность тоже субъективный фактор. Операции прямого доступа к структуре данных являются примитивными по оп-

ределению. Операции, которые можно свести к нескольким примитивным, но при этом затрачиваются значительные вычислительные ресурсы, также являются потенциально кандидатами на включение в разряд примитивных.

Эвристический подход к выбору операций

Функциональная семантика. Реализация интерфейса класса или модуля является сложной задачей. Обычно вначале делается первое приближение по структуре класса, а затем выявляются требования по уточнению, модификации и дополнению, обеспечивающие связь с пользователями этого класса. В частности может возникнуть потребность в создании новых классов или в изменении взаимодействия между существующими.

В пределах каждого класса принято иметь только простые (примитивные) операции, отражающие отдельные аспекты поведения. Такие методы называются тонко-структурированными. Принято также отделять методы, не связанные между собой. Это облегчает образование подклассов с переопределением логики поведения. Решение о количестве определяемых методов может быть обусловлено двумя причинами: описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода; расщепление метода усложняет интерфейс, но делает каждый из методов проще. По наблюдению Мейера «хороший проектировщик умеет найти компромисс между большим числом связей из-за дробления системы на фрагменты и большим размером модулей, еще поддающихся управлению» [53].

В OOD принято создавать методы для класса как целого, поскольку эти методы неразрывны в процессе реализации внутреннего протокола абстракции. Таким образом, определив характер поведения, нужно решить в каком из классов это поведение реализуется. Хальберд и О'Брайен предложили следующие критерии для принятия такого решения:

- * «Повторяемость» Будет ли это поведение реализовано в нескольких вариантах?
- * Сложность Как трудно реализовать такое поведение?
- * Применимость Насколько данное поведение характерно для конкретного класса?
- * Знание реализации Зависит ли реализация данного поведения от особенностей структуры класса?» [54]

Обычно операции декларируются в том классе, к объектам которого относятся данные действия. Однако в языках Object Pascal, C++, CLOS и Ada допускается описание операции в виде общедоступных подпрограмм, группируемых в утилиты класса. Общедоступная подпрограмма по терминологии C++ — это функция, не являющаяся элементом класса. Общедоступные подпрограммы не могут переопределяться как обычные методы и остаются неизменными. Наличие утилит класса позволяет выполнить требование простоты (примитивности) и уменьшить взаимосвязанность классов, особенно операций высокого уровня над объектами различных классов.

Распределение памяти и времени. После того как мы определились с функциональными вопросами, следует принять решение о распределении времени и памяти, которое соответствует выполняемым действиям. Для выражения таких решений принято использовать понятие лучшего, среднего и худшего вариантов, где худший — это нижний допустимый предел. К

данной проблеме тесно примыкает вопрос синхронизации, хотя для большинства языков программирования синхронизация просто не реализуется из-за одноканального управления (последовательности объектов).

В таком случае говорят о передаче сообщений, так как их семантика сходна с обычным вызовом подпрограмм. В языках, реализующих параллельность (таким как Ada), передача сообщений уже не является простым процессом, поскольку нужно решать вопросы неопределенности управления объектом по двум независимым каналам. Объекты, защищенные от такой неопределенности, являются либо заблокированными, либо параллельными (в зависимости от активизации самого объекта).

Для отражения параллельности отдельных операций и объекта в целом используются специальные семантические приемы. В соответствии с ними выделяют следующие формы передачи сообщений:

- * Синхронная Операция активизируется только по готовности передающего и принимающего сообщения; ожидание взаимной готовности может быть неопределенно долгим.
- * Отсроченная Соответствует в основном синхронной, но в случае неготовности принимающего передающий не выполняет операцию.
- * Задержанная Вид синхронной с оговоренным временем ожидания готовности принимающего.
- * Асинхронная Посылающий сообщение выполняет операцию вне зависимости от готовности принимающего.

Нужная форма выбирается для каждой операции только после принятия функциональных решений.

Эвристический подход к определению взаимоотношений

Критерии оценки взаимоотношений. Отношения между классами и объектами связаны с конкретными действиями. Если мы хотим, чтобы объект X послал объекту Y сообщение M, то прямо или косвенно класс X должен быть доступен («видим») для класса Y, иначе невозможно определить в классе X операцию M. Объект X также должен быть «видим» для Y, иначе Y не будет знать о существовании X. Под доступностью и видимостью понимается способность одной абстракции обращаться к внешним ресурсам другой абстракции. Таким образом, взаимозависимость является мерой видимости.

Одним из полезных правил является закон Деметера, который установил, что «методы любого класса не должны зависеть от структуры других классов, за исключением собственной структуры (верхнего уровня). В каждом методе посылаются сообщения только объектам из предельно ограниченного множества классов» [55]. Следование этому закону позволяет создавать близко связанные классы, реализация которых защищена от доступа. Такие классы достаточно автономны и для понимания их логики нет необходимости знать строение других классов. При анализе структуры классов для системы в целом можно обнаружить, что иерархия наследования либо широкая и мелкая, либо узкая и глубокая, либо сбалансированная.

В первом случае структура классов выглядит как лес из свободно стоящих классов, которые могут свободно смешиваться и вступать во взаимоотно-

ношения [56]. Во втором случае структура классов напоминает одно дерево из ветвей-классов, имеющих общего предка [57]. Каждый из вариантов имеет свои достоинства и недостатки. Между отдельными классами существует более тесная взаимозависимость, но совсем не используется общность структуры. В случае дерева классов эта общность используется максимально, поэтому каждый из классов имеет меньший размер.

Однако для понимания сущности таких классов нужно знать все особенности наследованных или использованных свойств других классов. Иногда требуется выбирать между отношениями наследования и использования. Например, следует ли в классе «автомобиль» использовать или наследовать классы «двигатель» и «колесо»? В данном случае более целесообразны отношения использования. По мнению Мейера, между классами А и В «отношения наследования более пригодны тогда, когда любой объект класса В может рассматриваться одновременно как объект А» [58]. Если же объект является больше, чем сумма отдельных частей, то более целесообразны отношения использования.

Роль механизмов реализации и «видимости». Отношения между объектами определяют в основном и механизмы их взаимодействия. Вопрос состоит только в направлении реализации определенных действий. Например, на ткацкой фабрике материалы (партии) поступают на участки для обработки. На каждом участке можно заменить управляющего. Является ли поступление материала на участок операцией над помещением, над материалом или тем и другим сразу? Если это операции над помещением, то помещение должно быть «видимо» для партии материала. Если это операция над материалом, то материал должен быть «видим» для помещения, так как партия материала должна различать помещения участков. В последнем варианте (операция над помещением и материалом) нужно обеспечить взаимную «видимость».

Теперь следует определить отношение между управляющим участком и помещением (но не материалом и управляющим); либо управляющий должен знать о помещении, либо помещение об управляющем. Иногда в процессе проектирования полезно определить в явном виде «видимость» объектов. Существуют три основных способа реализации видимости объекта X объекту Y:

- * Размещение в одной Y находится в зоне видимости X; поэтому X может прямо именовать Y
- * Использование Y передается в качестве параметра какой-либо операции над X
- * Использование поля Y является полем объекта X

Эти способы являются вариациями идеи общей зоны «видимости». Y может быть полем X и при этом находится в зоне видимости других объектов. В языке Smalltalk такой способ означает зависимость двух объектов. Общая зона видимости приводит к общности структуры, т.е. общая часть структуры доступна по нескольким направлениям. Такие отношения не всегда желательны, поэтому целесообразно пользоваться их явным указанием в процессе проектирования.

Эвристический подход к реализации выбора

Внутреннее строение (реализация) классов и объектов разрабатывается только после завершения проектирования их внешнего облика. При этом необходимо принять два проектных решения: выбрать способ представления класса или объекта и способ размещения их в модуле.

Представление классов и объектов. Представление классов и объектов почти всегда связано с ограничением доступа к элементам абстракции. Это позволяет вносить изменения (например, перераспределение памяти и временных ресурсов) без нарушения функциональных связей с другими классами и объектами. Вирт считает, что «выбор способа представления является нелегкой задачей и не определяется только техническими возможностями. Он всегда должен рассматриваться с точки зрения операций над данными» [59]. Рассмотрим, например, класс, соответствующий совокупности планов полета самолетов. Как его нужно оптимизировать — по эффективности поиска или по времени включения в план и удаления из него? Поскольку невозможно реализовать и то и другое одновременно, нужно сделать выбор на основе знаний и характера задачи. Не всегда удастся сделать такой выбор и тогда создастся семейство классов с одинаковым интерфейсом, поведение которых зависит от направления оптимизации.

Одним из наиболее трудных решений является выбор между возможностью вычисления элементов состояния объекта и их хранением в виде поля данных. Рассмотрим, например, класс «корпус» с соответствующим ему методом «объем». Этот метод возвращает значение объема объекта. В структуре объекта хранятся данные о высоте конуса и радиусе основания в виде отдельных полей. Следует ли еще создать поле данных для значения объема или следует вычислять его по мере необходимости с помощью метода «объем» [60]? Если мы хотим получать значение объема максимально быстро, нужно создавать соответствующее поле данных. Если важнее экономия памяти, лучше вычислить это значение. Оптимальный способ представления объекта всегда определяется характером решаемой задачи. В любом случае этот выбор не должен зависеть от внешних особенностей (интерфейса) класса, наоборот, такой выбор не должен сказываться на отношениях с объектами-пользователями.

Размещение классов и объектов в модуле программы. Аналогичный вопрос возникает при выборе места для декларирования классов и объектов в программном модуле. В языке Smalltalk эта проблема отсутствует, здесь модульный механизм не реализуется. В языках Object Pascal, C++, CLOS и Ada существует понятие модуля как отдельной языковой конструкции. Решение о месте декларирования классов и объектов в этих языках является компромиссом требований «видимости» и защиты информации. В общем случае модули должны быть функционально связанными и независимыми. При этом следует учитывать ряд истехнических факторов, таких, как повторное использование, документирование, требования секретности. Проектирование модулей — не менее простой процесс, чем проектирование классов и объектов. О проблеме защиты информации Парнас, Клементс и Вейс говорят следующее: «Реализация этого принципа не всегда является очевидной. Необходимо минимизировать стоимость программных средств (в целом за время эксплуатации) и оценить вероятность внесения изменений. Такая оценка исходит из практического опыта и знания предметной области, включая технологию программирования и аппаратную реализацию» [61].

Заключение

- * Объект характеризуется состоянием, поведением и индивидуальностью.
- * Структура и поведение схожих объектов описываются в общем для них классе.
- * Состояние объекта охватывает все его свойства (обычно статические) и их текущие значения (динамические).
- * Поведение объекта характеризует изменение его состояния в процессе взаимодействия с другими объектами и формируемые при этом сообщения.
- * Индивидуальность объекта — это свойство, отличающее его от всех других объектов.
- * Иерархия объектов может строиться на принципах использования или включения.
- * Множество объектов с одинаковой структурой и поведением является классом.
- * Иерархия классов может строиться на принципах наследования, использования, наполнения (конкретизации) и включать метаклассы.
- * Классы и объекты, образующие словарь предметной области, называются ключевыми абстракциями.
- * Структура, объединяющая множество объектов и обеспечивающая их совместное, целенаправленное функционирование, называется механизмом реализации.
- * Качество абстракций может быть оценено критериями взаимозависимости, связанности, достаточности, полноты и простоты.

Дополнительная литература

MacLennan [G, 1982] рассматривает различия между значениями и объектами. В работе Meyer [J, 1987] вводится идея программирования в виде взаимодействия. Статьи Albano [G, 1983], Brachman [J, 1983], Hailpern и Nguyen [G, 1987], Wegner и Zdonik [J, 1988] составляют прекрасный теоретический фундамент по всем вопросам данной главы. Cook и Palsberg [J, 1989] дали формальное описание наследования. Wirth [I, 1987] предложил расширение для типа «запись» в языке Oberon.

Ingalls [G, 1986] подробно рассматривает множественный полиморфизм. Практическое руководство по эффективному использованию механизма наследования предложено Meyer [G, 1988], Halberg и O'Brien [G, 1988]. LaLonde и Pugh [J, 1985] разработали подходы к обучению использования обобщения и специализации.

Meyer [G, 1986] рассмотрел вопросы обобщения и наследования применительно к языку Eiffel. Stroustrup [G, 1988] предложил механизм параметризации типов в C++.

Альтернативный подход к иерархии классов реализуется с помощью механизма делегирования экземплярами. Подробно этот механизм описан Stein [G, 1987].

Глава 4

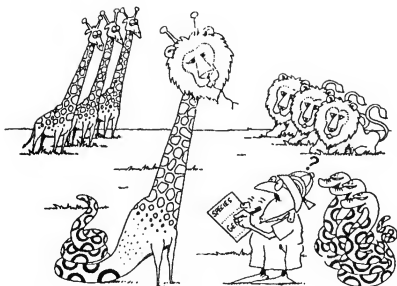
Классификация

Классификация — средство упорядочения знаний. В объектно-ориентированном анализе определение общих свойств объектов помогает найти общие ключевые абстракции и механизмы, что в свою очередь приводит к более простому проекту системы. К сожалению, пока не разработаны строгие методы классификации и нет правила, позволяющего выделять классы и объекты. Нет таких понятий, как «совершенная структура классов», «правильный выбор объектов». Как и во многих технических дисциплинах, выбор классов является компромиссным решением. На одной из конференций программистам был задан вопрос: «Какими правилами вы руководствуетесь при определении классов и объектов?» Страустрап, разработчик языка C++, ответил шуткой: «Мне помогает святой Грааль». Габриель, один из разработчиков CLOS, ответил: «Это вопрос, на который нет простого ответа. Я просто пробую» [1]. К счастью, имеется богатый опыт классификации в других областях науки, на основе которого разработаны методики объектно-ориентированного анализа и прикладного анализа. Каждая такая методика предлагает свои правила идентификации классов и объектов.

4.1. ВАЖНОСТЬ ПРАВИЛЬНОЙ КЛАССИФИКАЦИИ

Классификация и объектно-ориентированное проектирование

Определение классов и объектов — одна из самых сложных задач объектно-ориентированного проектирования. Успешному решению этой задачи обычно сопутствуют открытия и изобретения. С помощью открытий мы познаем ключевые понятия и механизмы, которые образуют "словарь" проблемы. С помощью изобретений мы конструируем обобщенные понятия, а также новые механизмы, которые определяют правила взаимодействия объектов. Поэтому открытия и изобретения являются неотъемлемой частью успешной классификации. Целью классификации является нахождение общих свойств объектов. Классифицируя, мы объединяем в одну группу объекты, имеющие одинаковое строение или одинаковое поведение. Разумная классификация, несомненно, часть любой точной науки. Михальский и Стер утверждают, что «неотъемлемой задачей науки является построение содержательной классификации наблюдаемых объектов и ситуаций. Такая классификация существенно облегчает понимание основной проблемы и дальнейшее развитие научной теории» [2]. Неудивительно, что классификация затрагивает многие аспекты объектно-ориентированного проектирования. Она помогает определить обобщенную, специализированную и собирательную иерархии классов. Определив общие формы взаимодействия объектов, мы найдем механизм, который может стать стержнем реализации проекта. Классификация помогает правильно определять модульную структуру. Мы можем расположить объекты в одном или разных модулях — это зависит от степени общности объектов; взаимосвязь и влияние — всего лишь меры этой общности. Классификация играет также роль при распределении процессов между процессорами, направляя процессы в один процессор или в разные процессоры в зависимости от того, как эти процессы связаны друг с другом.



Классификация упорядочивает знания.

Трудности классификации

Примеры классификаций. В гл. 3 мы определили «объект» как нечто, имеющее четкие границы. На самом деле это не совсем так. Границы предметов часто очень нечеткие. Например, посмотрите на вашу ногу. Попытайтесь определить, где начинается и кончается колено. На примере разговорной речи трудно понять, почему некоторым образом соединенные звуки определяют целое слово, а не часть какого-то еще большего слова. И если мы проектируем некоторую аудиосистему, стоит ли определять класс как состоящий из звуков или состоящий из слов? Как понимать отдельные фразы, предложения, параграфы, документы?

То, что разумная классификация — трудная проблема — факт, известный давно. И поскольку следует ожидать такие же трудности в объектно-ориентированном проектировании, рассмотрим примеры классификации в биологии и химии. Вплоть до 18 в. идея о возможности классификации живых организмов по степени сложности была господствующей. Мера сложности была субъективной, поэтому неудивительно, что человек оказался в списке на первом месте. В середине 17 в. шведский ботаник Карл Линней предложил более подробную таксономию для классификации организмов: он ввел понятия рода и вида. Век спустя Дарвин выдвинул теорию, по которой механизмом эволюции является естественный отбор и ныне существующие виды животных — продукт эволюции древних животных. Теория Дарвина основывалась на разумной классификации видов. Как утверждает Дарвин, «натуралисты пытаются расположить виды, роды, семейства в каждом классе с помощью натуральной системы отбора. Что подразумевается под этой системой? Некоторые авторы понимают некоторую простую схему, позволяющую

шую расположить наиболее похожие живые организмы в один класс и различные — в разные классы» [3]. В современной биологии термин «классификация» обозначает «установление иерархической системы категорий на основе некоторых предопределенных связей между организмами» [4]. Наиболее общее понятие в биологической таксономии — мир, затем в порядке убывания общности тип, класс, подтип, сорт, семейство, род и, наконец, вид. Исторически сложилось, что место каждого организма в иерархической системе определяется на основании внешнего и внутреннего строения тела и эволюционных связей. В современной классификации живых организмов определяются группы организмов, имеющих общее генетическое наследство, т.е. организмы, имеющие общие DNA, включаются в одну группу.

Возможно, для программиста-математика биология представляется зрелой, вполне сформировавшейся наукой с определенными критериями классификации организмов. Но это не так. По словам биолога Мзя: «На сегодняшний день мы даже не знаем порядок числа видов растений и животных, населяющих нашу планету: классифицировано менее чем 2 млн. видов, в то время как возможное число видов оценивается от 5 до 50 млн.». [5] Более того, различные критерии классификации одних и тех же организмов приводят к разным результатам. Мартин утверждает, что «все зависит от того, что вы хотите получить. Если вы хотите, чтобы классификация отражала генетическое родство видов, вы получите один ответ, если вы желаете получить информацию об адаптационных свойствах организмов — ответ будет другой» [6]. Можно заключить, что даже в строгих научных дисциплинах методы и критерии классификации сильно зависят от результата, который вы хотите достичь.

Аналогичная ситуация сложилась и в химии [7]. В древние времена считалось, что все вещества — суть комбинации земли, воздуха, огня и воды. В настоящее время такая классификация не может считаться сколько-нибудь удовлетворительной. В середине 1600-х годов химик Роберт Бойль предположил существование элементов как неких примитивных элементов, из которых составляются более сложные вещества. Век спустя химик Лавуазье опубликовал первый список, содержащий 23 элемента, хотя впоследствии было открыто, что некоторые из них не являются элементами. Но открытие новых элементов продолжалось, список увеличивался. Наконец, Менделеев предложил периодический закон, который давал точные критерии для классификации известных элементов и даже мог предсказывать свойства еще не открытых элементов. Но даже это открытие не оказалось окончательным решением задачи классификации элементов. В 1900-е годы были открыты элементы с одинаковыми химическими свойствами, но с разной атомной массой — изотопы. Как утверждал Декарт: «Открытие какого-либо порядка вещей не просто, но если он найден, нет никаких трудностей в его понимании» [8].

Методы классификации. Все это мы привели здесь не для того, чтобы оправдать длительность разработки программного обеспечения, хотя на самом деле многим менеджерам и пользователям кажется, что необходимы века, чтобы закончить начатую работу. Мы просто хотели подчеркнуть, что разумная классификация — работа интеллектуальная и лучший способ ее ведения — последовательный итеративный процесс. Это становится очевидно при анализе процесса разработки таких программных продуктов, как графический интерфейс, база данных и языки программирования четвертого поколения. Шоу утверждает, что «развитие какой-либо абстракции часто следует общему рисунку. В начале проблема решается ad hoc. По мере накопления

опыта некоторые решения оказываются более удачными, чем другие, развивается некоторый фольклор по данной теме. Удачные решения изучаются более систематично. Это позволяет развить модель, которая помогает определить способ реализации и разработать теорию, обобщающую это решение, что в свою очередь повышает уровень всей разработки и позволяет взяться за еще более сложную задачу, к которой в свою очередь мы подходим *ad hoc*, тем самым замыкая круг» [9].

Последовательный итеративный подход непосредственно определяет процедуру конструирования иерархий классов и объектов при разработке сложного программного обеспечения. На практике обычно за основу берется какая-то определенная структура классов, которую постепенно совершенствуют. И только на поздней стадии разработки, когда уже получен некоторый опыт использования такой структуры, мы можем критически оценить качество получившейся классификации. Основываясь на полученном опыте, мы можем создать новые подклассы из уже существующих или слить несколько существующих в один (декомпозиция). Возможно, в процессе разработки были найдены новые общие свойства, ранее не замеченные, и мы можем определить новые классы (*abstraction*) [10].

Почему же классификация так сложна? Мы объясняем это двумя важными причинами. Во-первых, потому, что нет определения «совершенная» классификация, хотя естественно некоторые лучше других. Комбс, Рафия и Фарол утверждают, что «существует столько способов деления мира на объектные системы, сколько ученых принимается за эту задачу: способ классификации определяется целью, к которой мы стремимся на этом пути» [11]. Флуд и Каркон приводят пример: «Соединенное Королевство экономисты могут рассматривать как экономическую систему, социологи — как социальное общество, правительство СССР — как военную угрозу и т.д.». [12] Во-вторых, разумная классификация требует большой творческой энергии и проницательности. Биртвисл, Дейх и Майтл заключают, что «иногда ответ очевиден, иногда он зависит от вкуса, а бывает, что выбор подходящих компонентов структуры очень критичен» [13]. Все это напоминает загадку — «Почему лазерный луч похож на золотую рыбку?..., потому что ни тот, ни другой не свистят» [14]. Только творческий разум может найти общее в столь разных и несвязанных предметах.

4.2. ИДЕНТИФИКАЦИЯ КЛАССОВ И ОБЪЕКТОВ

Классический и современный подходы

Еще со времен Платона проблема классификации занимала умы различных философов, лингвистов, когнитивистов, математиков. Поэтому было бы разумно изучить накопленный опыт и применить его по возможности в объектно-ориентированном проектировании. Исторически сложились только три основных подхода классификации:

- * Классическое распределение по категориям.
- * Концептуальное объединение.
- * Теория прототипов [15].

Классическое распределение по категориям. В классическом подходе распределения по категориям: «все вещи, обладающие данным свойством или совокупностью свойств, формируют некоторую категорию. Причем наличие этих свойств является необходимым и достаточным условием, определяющим категорию» [16]. Например, холостые люди определяют категорию: каждый

человек или холост, или женат этот признак достаточен для решения вопроса, к какой группе принадлежит тот или иной индивидуум. С другой стороны, высокие люди не определяют категорию, если, конечно, мы специально не уточним критерий, позволяющий четко различать высоких людей от невысоких. Впервые классическое распределение по категориям упоминается Платоном. В том виде, в котором им пользовался Аристотель, оно напоминает современную детскую игру в «двадцать вопросов» (это минерал, животное или растение? Это имеет мех или перья? Может ли оно летать?) [17]. Такой подход нашел последователей; наиболее выдающиеся из них: Акуинас, Декарт, Лок. По утверждению Акуинаса: «Мы имеем вещь согласно нашим знаниям об их свойствах» [18].

Принципы классического распределения по категориям четко проявляются в современной теории развития детей. Пойдет утверждать, что после первого года жизни ребенок осознает постоянство объектов и затем начинает приобретать навыки их классификации, сначала пользуясь базовыми категориями, такими, как собаки, кошки и игрушки [20]. Позднее ребенок осознает более общие категории, такие, как животные и более специфические (как колли, доги, гонимые) [21].

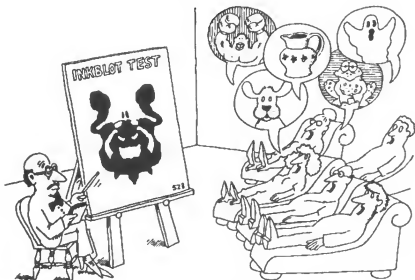
Проблемы классификации

На рис. 4-1 показаны 10 изображений поездов, пронумерованных от А до J. Каждое изображение состоит из поезда и нескольких вагонов. Прежде чем продолжать чтение, попытайтесь за 10 мин определить несколько групп изображений, составленных по какому-то логическому признаку. Например, изображения можно разбить на три группы: в одной группе поезда имеют черные колеса, в другой группе — белые колеса, в третьей — и белые, и черные.

Этот пример приводится из работ Степа и Михальского о концептуальном объединении [19]. Очевидно, что «правильного» разбиения на группы не существует. Наши изображения были классифицированы 93 различными способами. Наиболее распространенный способ классификаций по длине состава: были выделены три группы: составы с двумя, тремя и четырьмя вагонами. Второй по популярности вид классификации — классификация по признаку цвета колес поезда. Сорок из девяноста трех видов классификации были уникальными. В нашем эксперименте большинство опрошенных предлагали одну из двух наиболее популярных видов классификации (по длине состава и цвету колес поезда). Один опрошенный определил две группы изображений: в одной группе составы помечены буквами, нарисованными с помощью только прямых линий (А, Е, F, H и I), в другой — буквами с кривыми линиями. Это, действительно, пример не тривиального мышления.

Можно изменить условие. Представим, что круги обозначают груз с токсичными веществами, прямоугольники — лесоматериалы, все остальные знаки обозначают пассажиров. Попробуйте теперь классифицировать изображения (очевидно, что новые знания изменят тип классификации).

При таких дополнительных сведениях большинство опрошенных предлагали новую классификацию, основанную на признаке, — содержит состав токсичный груз или нет? В заключение можно сказать, что новые сведения об области применения позволяют представить более разумную классификацию.



Разные наблюдатели по-разному классифицируют один и тот же объект.

Суммируя сказанное выше, можно заключить, что для метода классического распределения по категориям наличие свойства является основным критерием схожести объектов. При этом объекты можно разделить на непересекающиеся множества, определив для каждого — обладает он конкретным свойством или нет. По предположению Мински: «Наиболее подходящий набор свойств для такой классификации характеризуется высокой независимостью этих свойств относительно друг друга. Это объясняет популярность такого набора свойств, как размер, цвет, форма и материал. Поскольку такие свойства независимы, мы можем применять их в любой комбинации» [22]. В более общем смысле свойства не обязательно могут определяться только измеряемыми характеристиками. Например, то, что птицы могут летать, а рыбы не могут, и есть то свойство, по которому можно отличить орлов от семги.

Конкретные свойства, которые необходимо выделить при классификации, определяются решаемой проблемой. Например, цвет автомобиля является важным свойством для какой-нибудь заводской системы контроля, но абсолютно неважен для программной системы, управляющей городскими светофорами. Вот почему мы утверждаем, что нет абсолютных критериев совершенства, хотя для конкретного примера некоторые структуры более пригодны, чем остальные. Джеймс утверждает, что «ни одна схема классификации не в состоянии представить структуру и порядок вещей в природе. Природа отнесется совершенно безразлично ко всем нашим методикам ее описания. Некоторые классификации будут более совершенными, чем другие, но только относительно наших интересов» [23].

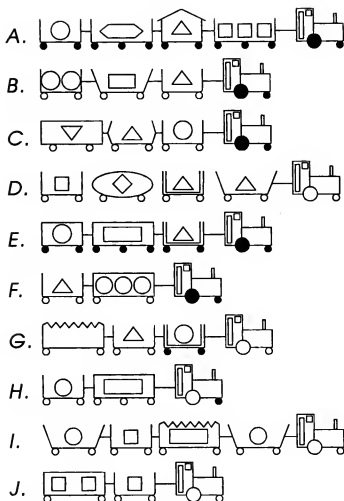


Рис. 4-1. Проблема классификации.

Концептуальное объединение. Концептуальное объединение — более современная вариация классического подхода. Она возникла из попыток формального представления знаний. Степ и Михальски утверждают, что при таком подходе сначала формируются концептуальные описания классов и затем объекты классифицируются согласно описанию, тем самым образуя классы [24]. Например, мы можем предложить концепцию «песня про любовь». Эта концепция непригодна для успешного классического распределения по категориям, поскольку трудно заключить — какая песня больше про любовь. Но тем не менее, если мы утверждаем, что песня скорее про любовь, чем про другое, то мы помещаем ее в категорию «песня про любовь». Та

кое распределение объектов по классам имеет явно выраженные вероятностные свойства.

Теория прототипирования. Классическое распределение по категориям и концептуальное объединение — достаточно мощные методы и вполне пригодные для проектирования сложных программных систем. Но все же есть ситуации, в которых эти методы не работают. Рассмотрим более современный метод классификации, теорию прототипов [25].

Существуют некоторые абстракции, которые не имеют ни четких свойств, ни четкого определения. По утверждению Витгейнштейна, существуют категории (например, игры), которые не соответствуют классическим образцам, так как нет признаков, свойственных всем играм. По этой причине их можно объединить так называемой семейственной схожестью. Витгейнштейн утверждает, что у категорий игр нет четкой границы. Категорию можно расширить и включить новые виды игр при условии, что они определенным образом соответствуют уже существующим играм [26]. При такой классификации класс определяется одним объектом-прототипом и объект можно включить в класс при условии, что он определенным образом похож на прототип.

Классификация и измененный объектно-ориентированный дизайн. Разработчику, озабоченному постоянно меняющимися требованиями к системе и страдающему от ограниченности ресурсов и необходимости выполнения насыщенных планов, предмет нашего обсуждения может показаться далеким от реальности. В действительности такой подход к классификации имеет непосредственное отношение к объектно-ориентированному проектированию. На практике мы идентифицируем классы и объекты исходя прежде всего из свойств рассматриваемой проблемной области. Подобные абстракции обычно легко можно выделить, так как они являются непосредственной частью словаря проблемной области [27]. Если с помощью этого подхода не удастся составить приемлемую структуру, приходится концептуально группировать объекты. Если теперь мы не сможем достаточно адекватно смоделировать задачу, тогда придется прибегнуть к классификации с помощью ассоциативных методов, выделяя группы объектов по признаку сходства их с некоторым объектом-прототипом. Эти три способа классификации составляют теоретическую основу объектно-ориентированного анализа групп и различных других методов, которые мы можем применить для идентификации классов и объектов при проектировании сложной программной системы.

Объектно-ориентированный анализ

Границы между стадиями анализа и проектирования размыты, но решаемые ими задачи определяются достаточно четко. В процессе объектно-ориентированного анализа мы моделируем проблему, определяя классы и объекты, которые формируют словарь проблемной области. При объектно-ориентированном проектировании мы находим абстракции и механизмы, обеспечивающие правильное поведение нашей модели. Таким образом, при разработке системы процесс проектирования продолжает работу, начатую на стадии анализа. Шклар и Меллор определили источники и кандидаты для классов и объектов [28]:

- * Материальные предметы Автомобили, телеметрические датчики, датчики давления
- * Роли Мать, учитель, полтник

Идея проведения прикладного анализа впервые была предложена Нейбором. Мы определим прикладной анализ как попытку выделить те объекты, операции, связи, которые эксперты данной области считают наиболее важными [31]. Муре и Бейлин определяют следующие этапы в прикладном анализе [32]:

- * «Построение каркаса модели после консультаций с экспертами.
- * Изучение существующих систем данной области и представление приобретенных знаний.
- * Определение схожести и различий между системами после консультаций с экспертами.
- * Пересмотр модели для описания существующих систем».

Прикладной анализ можно отнести к аналогичным применениям (вертикальный прикладной анализ) или к связанным частям одного и того же применения (горизонтальный прикладной анализ). Например, если мы начинаем проектировать систему учета пациентов, имеет смысл рассмотреть уже имеющиеся подобные системы, чтобы понять, какие ключевые абстракции и механизмы, использованные в них, будут полезны нам. Аналогично система отчета и документации должна представлять различные виды отчетов. Рассматривая отчеты как некоторую прикладную сферу, прикладной анализ может привести разработчика к пониманию ключевых абстракций и механизмов, которые обслуживают все виды отчетов. Полученные таким образом классы и объекты представляют собой множество ключевых абстракций и механизмов, отобранных с учетом исходной задачи создания системы отчетов. Поэтому окончательный проект будет проще.

Определим теперь, кто такой эксперт? В роли эксперта часто выступает просто пользователь системы, например инженер или диспетчер. Он необязательно должен быть программистом, но должен быть близко знаком с исследуемой проблемой и разговаривать на языке этой проблемы. Менеджеры проектов заинтересованы в непосредственном сотрудничестве пользователей и разработчиков системы. Но для очень сложных систем прикладной анализ является формальным процессом, для которого требуется большое число экспертов и разработчиков на длительный период времени.

На практике такой формальный анализ требуется редко. Обычно для начального выяснения проблемы достаточно короткой встречи экспертов и разработчиков. Удивительно, как мало информации требуется для продуктивной работы разработчика. Однако мы считаем чрезвычайно полезным такие встречи в течение всей разработки.

Альтернативные подходы

Объектно-ориентированный анализ и прикладной анализ — методы анализа, которые мы выбрали для объектно-ориентированного проектирования. Но существуют еще два альтернативных подхода, которые заслуживают обсуждения.

Метод неформального описания. Первая альтернатива — чрезвычайно простой метод, впервые предложенный Эбботом. По этому методу в описании проблемы подчеркиваются существительные и глаголы [33]. Существительные представляют собой кандидатов для классов, а глаголы — кандидатов для операций над классами. Метод можно автоматизировать, и такая система была построена в Токийском технологическом институте в Фиджитсу [34].

Подход Эббота полезен, так как он прост и заставляет разработчика сформировать словарь проблемы. Однако он непригоден для решения достаточно сложных проблем. Человеческий язык — очнь точное средство выражения, потому список объектов и операции зависит от умения разработчика записывать свои мысли. Тем более для многих существительных можно найти соответствующую глагольную форму, и наоборот.

Структурный анализ. Вторая альтернатива определения классов и объектов использует возможности структурного анализа для целей объектно-ориентированного проектирования. Такой подход привлекателен потому, что много аналитиков применяют этот подход и имеется много программных средств, поддерживающих структурный анализ.

После проведения структурного анализа мы уже имеем модель системы, описанную диаграммами потока данных и другими результатами структурного анализа. Эти диаграммы обеспечивают нас разумной формальной моделью проблемы. Исходя из этой модели, мы можем приступить к определению классов и объектов нашей проблемы тремя различными способами.

Макменамен и Палмер предлагают сначала приступить к формированию словаря данных и затем к анализу контекстных диаграмм модели. Они утверждают, что «рассматривая список основных структур базы данных, следует подумать, о чем они говорят или что описывают. Например, если они прилагательные, то какие существительные они описывают. Ответы на такие вопросы могут пополнить ваш словарь». Эти кандидаты для классов определяются из окружающей среды, входа и выхода системы, ее ресурсов.

Следующие два способа включают анализ диаграммы потоков данных, где кандидаты для объектов могут быть определены из следующих источников [36]:

- * Внешние события и объекты.
- * База данных.
- * Поток управления.
- * Преобразование потока управления.

Кандидаты для классов определяются из двух источников:

- * Потоки данных.
- * Поток управления.

Преобразование данных мы можем рассматривать как операции над существующими объектами или как поведение некоторого объекта, который мы специально создали для обоснования преобразования данных.

Сейдвич и Старк предлагают еще один метод, который они называют абстрактным анализом. Метод базируется на идентификации основных сущностей, которая по природе аналогична основным преобразованиям в структурном анализе. «В структурном анализе входные и выходные данные изучаются до тех пор, пока они не достигнут высшего уровня абстракции. Процесс преобразования входных данных в выходные есть основное преобразование. В абстрактном анализе разработчик делает то же самое, а также изучает основное преобразование для того, чтобы определить, какие процессы и состояния представляют наилучшую абстрактную модель системы» [37]. После определения основной сущности в диаграмме потоков данных абстрактный аналитик приступает к изучению всех поддерживающих систем, прослеживая входящие и исходящие потоки данных из центра, группируя процессы и состояния, встречающиеся по пути. Сейдвич и Старк нашли абстрактный анализ слишком сложным и как альтернативный предлагают объектно-ориентированный анализ [38].

Необходимо отметить, что принципы структурного проектирования и анализа ортогональны принципам объектно-ориентированного проектирования. Наш опыт показывает, что структурный анализ может быть полезен в процессе объектно-ориентированного проектирования, но при условии, что разработчик не отдаст предпочтение структурному типу мышления. Другая опасность анализа заключается в том, что многие аналитики любят рисовать диаграммы потоков данных, которые отражают скорее проект, чем модель. Очень трудно построить объектно-ориентированную модель, если она легко и очевидным образом поддается алгоритмической декомпозиции. Поэтому мы предпочитаем объектно-ориентированный анализ и анализ проблемной области как подготовительный этап для объектно-ориентированного проектирования. При этом уменьшается риск испортить проект элементами алгоритмического анализа. Если же необходимо применить структурный анализ, не следует писать диаграммы потоков данных, если они начинают представлять собой проект системы, а не существенную часть модели. Разумно избегать элементов структурного анализа, когда проектирование находится на последнем этапе. Необходимо также понимать, что различные компоненты проекта, полученные на этапе разработки (такие, как диаграммы потоков), не представляют собой какого-то конечного продукта, а всего лишь промежуточное средство, необходимое для понимания разработчиком проблемы. Обычно пишутся диаграммы потоков данных, а затем разрабатываются механизмы, обеспечивающие необходимое поведение системы, т.е. сам акт проектирования видоизменяет начальную модель. Поддержка модели в соответствии с проектом — работа рутинная и трудная, к тому же вряд ли необходимая. Таким образом, может сохраниться только продукт структурного анализа высокого уровня абстракции. Он охватывает модель проблемы и может служить основой для многих проектов.

4.3. КЛЮЧЕВЫЕ АБСТРАКЦИИ И МЕХАНИЗМЫ

Определение ключевых абстракций

Нахождение ключевых абстракций. *Ключевые абстракции* — это класс или объект, который определяет часть словаря проблемной области. Самая главная роль ключевых абстракций заключена в том, что они определяют границы нашей проблемы: выделяют вещи, существующие в нашей системе и поэтому важные для нас, и затеняют моменты системы, которые излишни. Задача выделения таких абстракций — задача, специфичная для каждой проблемной области. Как утверждает Голбери: «Соответствующий выбор объектов зависит от области приложения и точности представления информации» [39].

Определение ключевых абстракций включает в себя два процесса: открытие и изобретение. Мы распознаем абстракции, используемые специалистами по предметной области; если эксперты используют эту абстракцию в своей речи, тогда она важна [40]. Изобретая, мы создаем новые классы и объекты, не являющиеся существенной частью предметной области, но полезные инструменты при реализации проекта. Например, пользователь автоматического секретаря применяет термины «отчеты», «депозиты», «изъятия»; эти термины — часть словаря предметной области. Разработчик такой системы использует те же абстракции, но вводит и свои, такие, как база данных, экранный диспетчер и т.д. — ключевые абстракции проектирования, а не предметной области.

Наиболее мощный способ определения ключевых абстракций — найти в проблеме ключевые абстракции, аналогичные существующим классам и объектам. Так как это проблема классификации, мы можем использовать классические и современные способы классификации, описанные выше.

Пересмотр ключевых абстракций. Определив некоторый набор кандидатов для ключевых абстракций, мы должны отобрать лучшие по критериям, описанным в предыдущих главах. По словам Страустрапа: «Часто это означает, что программист должен задаваться вопросами: Как создаются объекты класса? Можно ли копировать или уничтожать объекты данного класса? Какие операции могут быть выполнены над этим объектом? Если ответы на эти вопросы туманны, то возможно общая концепция не точна и лучше еще раз пересмотреть, прежде чем заняться реализацией проекта» [41].

Определив новые абстракции, мы должны найти их место в контексте уже существующих классов и объектов. Подобное определение не имеет четко выраженного стиля, такого, как при проектировании сверху вниз или снизу вверх. Халберт и О'Брайен утверждают, что «ист. особой необходимости строить иерархию классов, начиная с самого верхнего класса, и потом дополнить подклассами. Чаще вы создаете несколько независимых классов, осознаете их общие черты и выделяете их в нескольких суперклассах. Несколько таких проходов вверх и вниз по иерархии вполне достаточно для создания программного проекта» [42]. Это замечание — не рекомендация для действий, а всего лишь наблюдение, основанное на опыте и подтверждающее тот факт, что объектно-ориентированное проектирование — процесс последовательный и итеративный.

Трудно сразу расположить классы и объекты на правильном уровне абстракции. Иногда мы находим в готовой классификации похожие подклассы и можем их передвинуть вверх в иерархии классов, таким образом увеличивая степень разделяемости общих свойств [43]. Аналогично можем найти классы со свойствами настолько общими, что происходит семантический разрыв между ними и их подклассами [44]. В обоих случаях мы пытаемся найти зависимые и независимые абстракции.

Программисты часто легкомысленно относятся к правильному наименованию классов и объектов, но на самом деле очень важно уловить в обозначении классов и объектов сущность описываемых ими предметов. Программы необходимо писать тщательно, как английскую прозу, не забывая читателей и компьютер. Рассмотрим имена, необходимые для идентификации одного объекта, — имя объекта, имя его класса, имя модуля, в котором этот класс объявлен. Учитывая, что таких объектов может быть тысяча, возникает проблема именования объектов.

Мы предлагаем следующие правила:

- * Объекты называются подходящими фразами:

TheSensor или *AShape*.

- * Классы называются общими существительными:

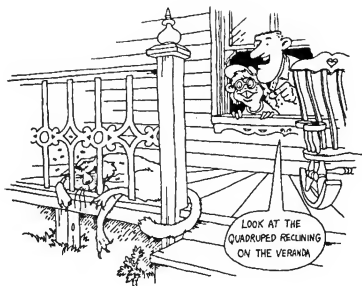
Sensors, Shapes.

- * Операции-модификаторы определяются соответствующими глаголами:

Draw, Move.

- * Операции-селекторы определяются вопросом или глагольной формой глагола «to be»:

ExtentOf, isOpen.



Классы и объекты должны соответствовать своему уровню абстракции.

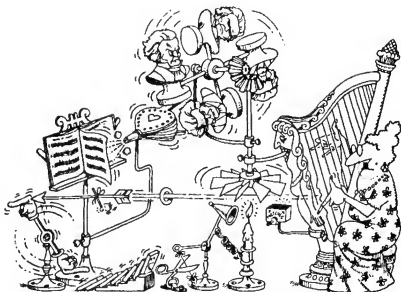
Определение механизмов

Нахождение механизмов. При разработке системы Джексона (Jackson System Development) [45] сначала были найдены ключевые абстракции, описывающие модель реальности, и только затем добавлены механизмы поведения [46]. Как уже говорилось выше, мы используем термин «механизм», чтобы описать некоторую структуру, с помощью которой объекты взаимодействуют между собой, обеспечивая необходимое поведение системы. Проект классов воплощает в себе знания о поведении классов. Проектирование механизмов соответствует решению задачи взаимодействия объектов.

Например, рассмотрим требование, предъявляемое к автомобилю: нажатие на акселератор должно приводить к увеличению оборотов двигателя. Как это происходит, водителю совершенно неинтересно. Для реализации требования можно применить любой механизм и его выбор — дело вкуса разработчика. Например, можно выбрать любой из предложенных ниже инженерных решений:

- * Механическая связь между акселератором и карбюратором (обычное решение).
- * Электронная связь датчика давления, находящегося под педалью акселератора и карбюратора.
- * Бак с горючим находится на крыше автомобиля и топливо свободно течет в двигатель. Поток топлива регулируется специальной заслонкой. Нажатие на педаль акселератора перекрывает заслонкой поток топлива в двигатель.

Какую именно реализацию выберет разработчик, зависит от таких параметров, как стоимость, надежность, технологичность и т.д.



С помощью механизмов объекты взаимодействуют друг с другом и обеспечивают таким образом более высокий уровень поведения системы.

Пользователь не должен нарушать правила пользования устройством, также и объекты должны вести себя в соответствии с реализацией их механизмов. Водитель был бы удивлен, если бы, нажав на педаль акселератора, он включил фары.

Ключевые абстракции определяют словарь проблемной области, механизмы определяют суть проекта. В процессе проектирования разработчик должен принимать во внимание не только структуру классов, но и то, как объекты этих классов взаимодействуют друг с другом. Если программист разработал какой-то конкретный механизм взаимодействия, он реализуется определением методов для объектов соответствующих классов.

Определение механизма представляет собой стратегическое решение. Аналогично проектирование структуры классов тоже стратегическое решение в противоположность проектированию интерфейса какого-то одного класса.

Стратегические решения должны проводиться в проекте явно и четко, иначе мы можем получить большое число независимых объектов, выполняющих свои функции без учета окружающих объектов. В наиболее элегантных быстрых программах воплощены тщательно разработанные механизмы.

Примеры механизмов. Рассмотрим механизм рисования, обычно применяющийся в графических интерфейсах пользователя. Для того чтобы представить какой-либо рисунок на экране, пользователю необходимо обеспечить взаимодействие ряда объектов просматриваемой модели и некоторого клиента, который знает когда (но не как) отображать на экране эту модель. Сначала клиент дает окну команду нарисовать себя. Так как окно может

включать в себя ряд подокон, оно в свою очередь приказывает каждому из них нарисовать себя. Каждое подокно посылает далее сообщение своей модели нарисовать себя, в результате чего и появляется изображение на экране. В этом механизме модель полностью отделена от окна и подокна, в котором оно представлено: подокна могут посылать сообщения к моделям, но модели не могут посылать сообщения подокнам. Smalltalk использует вариант этого механизма, названный модель рисунок контроллер (МРК) [47].

Механизмы, таким образом, представляют другой уровень повторного использования в проектировании более высокий, чем повторное использование индивидуальных классов. Метод МРК, например, является основой интерфейса пользователя в языке Smalltalk. Этот метод строится на механизме зависимостей, которым обладает базовый класс языка Smalltalk, класс Object и который, таким образом, часто используется библиотекой классов языка Smalltalk.

Примеры механизмов можно найти во многих системах. Структуру операционной системы, например, можно описать на высоком уровне абстракции по тем механизмам, которые используются для диспетчеризации программ. Система может быть монолитной (как MS-DOS), организованной как наращиваемое ядро (UNIX) или как иерархия процессов (операционная система THE) [48]. В системах с искусственным интеллектом используются разнообразные механизмы принятия решений. Одним из наиболее распространенных является механизм рабочей области, в котором каждый индивидуальный источник знаний независимо изменяет рабочую область. В таком механизме не существует центрального контроля, но любое изменение в рабочей области может явиться толчком для разработки системой нового пути решения поставленной задачи [49].

На этом завершается наше изучение классификации и понятий, являющихся основой объектно-ориентированного проектирования. Следующие три главы посвящены самому методу, в частности системе обозначений, процессу проектирования и рассмотрению практических примеров.

Заключение

- * Идентификация классов и объектов — одна из важных задач объектно-ориентированного проектирования.
- * Классификация — в основном есть проблема группирования объектов.
- * Классификация — последовательный и итеративный процесс; трудности классификации обусловлены в основном широким выбором возможных решений.
- * Три подхода классификации включают классическое распределение по категориям (группирование по свойствам), концептуальное объединение (группирование по некоторой концепции) и теорию прототипов (группирование объектов по некоторым признакам схожести с прототипом).
- * Кандидаты для классов и объектов берутся из реальных объектов, событий, возможных ролей и взаимодействий.
- * Прикладной анализ определяет классы и объекты по свойству их присутствия в различных применениях в данной прикладной области.
- * Ключевые абстракции отражают словарь проблемы.
- * Механизмы — существенный момент проекта; и определяют взаимодействия различных объектов системы.

Дополнительная литература

Проблема классификации вечна. В работе «Statesman» Платон описывает метод классификации, группируя объекты с одинаковыми свойствами. В работе «Категории» Аристотель затрагивает эту же тему и анализирует различия между классами и объектами. Несколько веков спустя Акуинас в работе «Summa Theologica» и Декарт в «Rules for the Direction of the Mind» также рассматривают философские вопросы классификации.

Классификация — присущая человеку способность. Теории, касающиеся способности человека классифицировать в детском возрасте, рассмотрены Piaget и Maier [A, 1969]. Lefrancois [A, 1977] предлагает вполне доступное введение в этот вопрос и вполне понятную теорию восприятия ребенком концепции объектов.

Многие ученые-когнитивисты подробно исследовали проблемы классификации. Hewell и Simon [A, 1972] предлагают оригинальный материал по этой теме. Информацию по этому вопросу можно получить из работ у Simon [A, 1982], Hofstadter [I, 1979], Siegler и Richards [A, 1982] и Stillings, Feinstein, Garfield, Rissland, Rosenbaum, Weisler и Baker-Ward [A, 1987]. Lakoff [A, 1987], будучи лингвистом, изучает вопрос влияния трудностей, связанных с классификацией на развитие человеческой лексики, и показывает, как эти исследования объединяют некоторые вопросы, связанные с человеческим мозгом. Minsky [A, 1986] подходит к этому вопросу с другой стороны, начиная с исследования структуры мозга.

Когнитивисты используют термин «концептуальное группирование», пользуясь им для представления знаний через классификацию. Концептуальное группирование подробно рассмотрено в работах Michalski и Stepp [A, 1983, 1986], Pckham и Sowa [A, 1984].

Проблемный анализ является средством нахождения ключевых абстракций и механизмом определения словаря проблемной области. Iscoe [B, 1988] внес большой вклад в эту область. Дополнительная информация может быть найдена в работах Iscoe, Browne, и Weth [B, 1989], Moore и Bailin [B, 1988], и Arango [B, 1989].

Определение классов и объектов может последовать после нахождения ряда различных моделей, которые удовлетворяют требованиям проблемной области. Abbott [F, 1983] предлагает начать поиск классов, проведя текстовое описание проблемной области. Ward [B, 1989] и Сайдевик (Seidewitz) и Stark [F, 1986] предлагают начать со структурного анализа диаграмм потока данных. Veryard [B, 1984] изучает эту проблему путем моделирования данных.

Математики пытались спроектировать новый подход к классификации, подводя ее близко к так называемой теории меры. Stevens [A, 1946] и Coombs, Raiffa и Thrall [A, 1954] предлагают работы по этому направлению.

Classification Society of North America издает два журнала в год, которые содержат статьи, освещающие проблемы классификации.

Часть II

МЕТОДОЛОГИЯ

Нельзя заранее предсказать, какие нововведения будут успешными, а какие — неудачными. Решив спроектировать что-то новое (мост, самолет или небоскреб), инженер стоит перед необходимостью выбора из множества альтернатив. Возможно, он решит воплотить в своем проекте все лучшие черты прошлых проектов, но, возможно, попытается улучшить некоторые характеристики системы.

Геири Петроски (Henry Petroski)
To Engineer is Human

Глава 5

Система обозначений

Проектирование — это не рисование диаграмм; диаграммы — всего лишь зарисовка проекта. Если вы понаблюдаете за работой инженера — программиста, механика, химика, архитектора, — вы скоро поймете, что сам проект находится в голове разработчика, и только иногда фрагменты проекта записываются на листочках бумаги [1].

Тем не менее хорошо определенная система обозначений имеет важное значение. Во-первых, стандартное описание проекта понятно другим разработчикам, и процесс обсуждения проекта упрощается. Электрические схемы символ транзистора, понятны электротехникам всего мира. Аналогично проект жилого дома, разработанный архитекторами в Нью-Йорке, будет понятен строителям в Сан-Франциско. Во-вторых, «хорошая нотация освобождает ум от рутинной работы и позволяет сконцентрироваться на более сложных и необходимых работах» [2]. В-третьих, четкая система обозначений может освободить разработчика от рутинной проверки на корректность и состоятельность — эту работу выполняют автоматические устройства. Разработка программного обеспечения всегда будет трудоемкой работой. «Машины будут выполнять «черную» работу, а разработка концепций развития — прерогатива человека. Нельзя автоматизировать проектирование концепций структуры, но можно упростить работу описания этой структуры» [3].

5.1. ЭЛЕМЕНТЫ СИСТЕМЫ ОБОЗНАЧЕНИЙ

Необходимость различных точек зрения

Нельзя представить все детали сложной программной системы в диаграмме одного типа. Необходимо понимать как функциональные, так и структурные свойства объектов. «Необходимо понимать таксонометрическую структуру класса объектов, используемые механизмы наследования, индивидуальное поведение объектов и динамическое поведение системы в целом.



Рис. 5-1. Модель объектно-ориентированного проектирования.

Задача в чем-то аналогична показу футбольного или теннисного матча, когда требуется много камер, расположенных в разных местах спортивной площадки. И каждая камера передает свой аспект игры, недоступный другим камерам» [4].

На рис. 5-1 представлены различные типы моделей (описанные в гл 1), которые, как мы считаем, могут оказаться необходимыми при объектно-ориентированном проектировании. Эти модели в совокупности позволяют разработчику записать все интересные решения; они достаточно полны, чтобы обеспечить разработку проекта на каком-либо объектно-ориентированном или объектном языке. Такая система обозначений пригодна для разработок программ в сотни и миллионы строк.

Наша система обозначений — довольно подробная, но это не означает, что каждый ее аспект необходимо использовать. При проектировании (например, в архитектуре) часто пользуются грубыми рисунками, и, только после того как закончена творческая созидательная часть проекта, разработчики пытаются подробно описать свой проект» [5]. Следует помнить, что графическая нотация — это средство документации проекта и не является завершающим этапом. Существует опасность слишком сильной спецификации отдельных решений, что может затруднить решение всей проблемы. Например, архитектор на эскизе может показать расположение выключателя, но точное его расположение может быть установлено только электриком после тщательного осмотра здания. Если разработчики и исполнители — высококвалифицированные специалисты и они установили тесный деловой контакт, то для работы, несомненно, достаточно наброска проекта. Если исполнители не имеют высокой квалификации и находятся далеко от разработчиков, необходим более детальный эскиз проекта. Часто в программных языках используются разные термины для описания одного и того же понятия. Система обозначений, которую мы здесь рассматриваем, не зависит от конкретных языков программирования. Конечно, некоторые элементы этой системы могут не иметь аналогов в языке, который по ряду причин необходимо использовать при реализации проекта. В этом случае не надо пользоваться этими элементами. Например, в Smalltalk нельзя объявить свободную подпрограмму, поэтому не следует использовать утилиты классов при описании проекта.

Ниже описаны синтаксис и семантика системы обозначений объектно-ориентированного проектирования. В качестве примера такой системы мы будем использовать тепличное хозяйство на основе гидропоники (гл. 2). Мы здесь не объясняем, как выбираем фигуры. Эти вопросы рассмотрены в гл. 6 и 7. В гл. 8-12 описаны применения такой системы обозначений.

Логическая и физическая модели

При разработке обозначений мы нашли необходимым четко разделить независимые стороны проектных решений. Разработчик должен уделить внимание следующим вопросам в объектно-ориентированном проектировании:

- * Какие классы и как они связаны между собой?
- * Какие механизмы обеспечивают взаимодействие объектов?
- * В каком месте необходимо объявлять классы и объекты?
- * Какому процессору приписать конкретный процесс, как управлять процессами?

Ответы на эти вопросы можно представить в виде следующих четырех диаграмм:

- * диаграмма класса.
- * диаграмма объектов.
- * диаграмма модулей.
- * диаграмма процессов.

Эти четыре диаграммы составляют основу системы обозначений объектно-ориентированного проектирования.

Первые две диаграммы — часть логического представления системы, потому что они служат для описания ключевых абстракций проекта. Последние две диаграммы — часть физической структуры системы, потому что они описывают конкретные программные и аппаратные компоненты реализации проекта.

Статическая и динамическая модели

Четыре диаграммы являются статическими. Программные системы по сути являются динамическими, так как в программе объекты создаются и уничтожаются, посылают друг другу сообщения. Описание динамической системы с помощью статических рисунков — задача сложная, но она возникает в каждой научной дисциплине. В объектно-ориентированном проектировании для описания динамической компоненты мы будем пользоваться еще двумя дополнительными диаграммами:

- * Диаграммы переходных состояний.
- * Временные диаграммы.

Каждый класс может иметь диаграмму переходных состояний, которая описывается как временная последовательность внешних событий, влияющих на объекты этого класса. Отдельная диаграмма объектов представляет собой фотоснимок текущих событий или изменяющейся конфигурации объектов. Поэтому необходимо вместе с диаграммой объектов описывать временную диаграмму системы, в которой представлен временной порядок сообщений и событий. При некоторых обстоятельствах структурированный англоязычный язык или выразительный PDL — вполне достойная замена для временной диаграммы. Кроме того, временная диаграмма или PDL могут быть использованы для документации динамического развития процесса.

Роль автоматизированных систем проектирования

Плохой разработчик, обеспеченный системой автоматического проектирования, может всего лишь создать плохой проект за более короткий срок. Хороший проект разрабатывается хорошим разработчиком, а не сложными автоматизированными системами. Автоматизированные системы освобождают разработчика от трудоемких процессов, позволяя сконцентрировать внимание на сложных аспектах проблемы.

Итак, есть задачи, которые автоматизированные системы решают хорошо, и есть задачи, которые они не способны решить в принципе. Например, если мы на диаграмме объектов показали, что один объект посылает сообщение другому объекту, автоматизированная система может подтвердить, что это сообщение является частью протокола между объектами. Это пример автоматического контроля. Когда мы используем в диаграмме объектов определенные обозначения, для того, чтобы показать принадлежность объектов классам или выразить мысль, что каждый объект диаграммы есть экземпляр определенного класса, мы можем надеяться, что автоматизированная система успешно использует такие обозначения.

Аналогично автоматизированная система может определить, что некоторые объекты класса никогда не используются. Это пример автоматического контроля. Более сложная автоматизированная система может определить длительность конкретной операции или потенциальный дедлок между одновременно активными объектами. Это пример автоматического анализа. Однако автоматизированная система не в состоянии определить новый класс, чтобы упростить нашу структуру классов. Мы, конечно, можем, используя экспертные системы, попытаться построить такую автоматизированную систему, но для этого требуются, во-первых, эксперты как в области объектно-ориентированного проектирования, так и в прикладной области, и, во-вторых, четко определить правила теории обучения умению классифицировать и способности накапливать общие знания.

5.2. ДИАГРАММА КЛАССОВ

Классы, взаимосвязь классов, утилиты классов

Диаграмма классов показывает классы и их иерархию, тем самым представляя логическую сторону проекта. Отдельная диаграмма классов представляет всю или часть структуры классов системы. Для небольших систем достаточно одной диаграммы классов, для больших систем потребуется ряд таких диаграмм.

Три наиболее важные компоненты структуры классов — классы, иерархия классов, утилиты классов (для языков, в которых определены свободные подпрограммы).

Классы. На рис. 5-2 показано обозначение для представления классов на диаграмме. Класс обычно представляют аморфным объектом, называемым облаком. Так обозначают не четко описанные абстракции. Штриховая линия, образующая границу, указывает, что обычно пользуются экземплярами этого класса, а не самим классом. Имя класса (name) помещается внутри облака. Если имя класса слишком длинное, его можно сократить. Каждый класс должен иметь уникальное имя.

Иерархия классов. В гл. 3 были описаны типы различных связей между классами: наследование, связь по типу использования, наполнение и

связь между метаклассами. Для усиления концепции иерархии классов мы использовали понятие «неопределенное взаимодействие». Обозначения для каждого типа связи между классами представлены на рис. 5-3. Каждое такое обозначение может иметь метку (не обязательно) для обозначения имени или роли этой связи. Если метка указана, можно использовать жирную точку для указания направления чтения метки. Связь по типу использования обозначается двойной линией с кружком на одном конце для обозначения класса, который пользуется ресурсами другого. Например, если мы используем двойную линию, для того чтобы указать на связь между классами А и В и помещаем незакрашенный кружок у класса А, то утверждаем, что интерфейс класса А использует ресурсы класса В. Если кружок закрашен, то утверждаем, что реализация класса А использует ресурсы класса В. На ранних стадиях разработки достаточно показать связь между интерфейсами, при реализации проекта можно будет добавить связь между реализациями классов.

При некоторых обстоятельствах, особенно при моделировании классов для базы данных, большое значение приобретают количественные отношения между классами, которые используют ресурсы друг друга. На рис. 5-3 показаны обозначения, которые мы используем для обозначения количественных отношений между классами (обозначения заимствованы из *gper* утилит OS UNIX). Например, между классами А и В, если мы поместим «1» у класса А и « ∞ » у класса В, то тем самым мы утверждаем, что для каждого экземпляра класса А имеем несколько экземпляров класса В, и для каждого экземпляра класса В — один экземпляр класса А. Если необходимо, мы можем сформулировать более сложные количественные отношения между классами, используя знаки «=», «>», «<», « \leq » и « \geq ».

Для обозначения того, что класс А наполняет класс В, используется штриховая линия, стрелка указывает на класс, который иллюстрируют (например, В). Такая связь классов используется только для языков, которые поддерживают параметризованные классы или обобщенные механизмы.

Наследственная связь обозначается так же, как и наполняющая, но линия сплошная. Наследственная связь наиболее популярная после связи использования. Обычно в иерархии классов не указывают самый верхний класс, такой, как TObject в Object Pascal, если, конечно, нет особой причины. Независимые классы обозначаются без указания их принадлежности множеству подклассов самого верхнего класса. Если экземпляры одного класса несовместимы по типу с экземплярами суперкласса этого класса, то стрелку, соединяющую эти классы, помечают черточкой.

Связь между метаклассами обозначается жирной линией. В некоторых языках, например в Smalltalk, каждый класс имеет свой метакласс, но мы обычно обозначаем только те метаклассы, которые определяют поведение, свойственное области приложения. Неопределенная связь между классами обозначается линией из точек. Штриховая линия применяется, если связь существует, но решение о типе связи отложено.



Рис. 5-2. Обозначение класса.



Рис. 5-3. Обозначение отношений между классами и множественностью классов.

Утилиты класса. Обозначение утилиты класса представлено на рис. 5-4. Оно похоже на обозначение класса, но отличается тенью. Утилиты класса являются общедоступными подпрограммами или множеством таких подпрограмм. Если программный язык не использует такие подпрограммы, такое обозначение можно не применять. Необходимо именовать утилиты, причем имена должны быть уникальными.



Рис. 5-4. Обозначение утилиты классов.

Пример диаграммы класса. На рис. 5-5 приведен пример обозначений тепличного хозяйства на основе гидропоники. Это лишь одна из многих диаграмм, необходимых для полного описания такой системы. На ней показано, что классы Heater (нагреватель) и Cooler (холодильник) наследуют из более общего класса Actuator. В реализации класса EnvironmentalController используются классы Heater, Cooler и Lights. Для каждого нагревателя и холодильника существует только один экземпляр класса EnvironmentalController, и для каждого экземпляра класса EnvironmentalController имеются только один нагреватель и холодильник.

Для экземпляра класса EnvironmentalController может существовать n объектов класса Lights, но для каждого экземпляра Lights — только один экземпляр EnvironmentalController. На диаграмме также показан модуль свободных подпрограмм, использующих ресурсы класса EnvironmentalController.

Категории классов и видимость категорий классов

Категории классов. Обычно диаграмма классов включает один или несколько десятков классов. Но большая система может содержать сотни и тысячи классов их невозможно поместить на одной диаграмме. Чтобы обойти подобные затруднения, необходимы средства описания подмножеств классов. Таким образом, мы приходим к понятию категория классов.

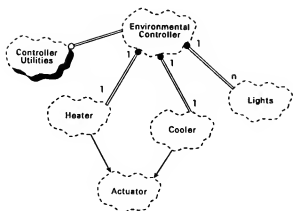


Рис. 5-5. Диаграмма классов тепличного хозяйства на основе гидропоники.

В немногих объектно-ориентированных языках классы определяются внутри других классов. Такая структурность редко отражает какие-то проектные решения, а всего лишь обеспечивает удобство реализации или ограничивает область видимости вложенных классов. Очень важным моментом проектирования является определение категорий классов (обычно известных как *subject areas*), каждая из которых представляет собой множество классов или других категорий классов, объединенных логическими средствами. Большинство объектно-ориентированных языков не имеют возможности своими средствами обеспечить выделение категории, но эта процедура является существенной частью проектирования. На практике большая система имеет диаграмму классов, состоящую из категорий классов с максимальным уровнем абстракции. Такая диаграмма позволяет разработчику понять общую логическую структуру системы, поскольку она описывает высокоуровневую организацию ключевых понятий, формирующих словарь проблемной области. Каждая категория классов определяет некоторую диаграмму классов; рассмотрев реализацию отдельной категории, мы увидим или просто диаграмму классов, состоящую из классов, связей, утилит, или в случае большой системы другую диаграмму категорий классов.

Обозначения категорий показаны на рис. 5-6: прямоугольник с именем в центре. Правила именования такие же, как для классов и утилит. Как мы уже говорили, категория включает другую диаграмму классов или категорий. Диаграмма классов может содержать как категории классов, так и просто классы.

Видимость категорий классов. Категория классов представляет собой именованную область, содержащую некоторые объекты: объекты могут быть видимыми извне, могут быть обособленными для категории классов, некоторые объекты могут быть импортированы из других категорий классов. Для обозначения видимости именованного объекта категории классов будем пользоваться обозначениями, показанными на рис. 5-6. Объект, имя которого не помечено, считается обособленным для данной категории, и поэтому на этот

объект нельзя ссылаться извне. Объект, имя которого обведено прямоугольником, хоть и содержится в данной категории, но может быть экспортирован, поэтому считается видимым для категорий, которые его импортируют. Например, предположим, что класс А экспортирован из категории X. Если категория классов X видима категории классов Y, то класс А видим всем объектам, содержащимся в классе Y. Другими словами, А импортируется в Y. Класс А определяется в категории X, но может появиться и в диаграммах классов для категории Y, причем необходимо подчеркнуть имя класса А в таких диаграммах.

Взаимосвязь между категориями классов обозначается стрелкой, показанной на рис. 5-6. Для того чтобы показать, что категория Y импортирует классы, определенные в X, мы проводим стрелки от Y к X. Каждая такая стрелка может обозначаться именем, чтобы документировать имя и роль связи.

При обозначении связи категорий с основными классами возникают трудности, обусловленные тем, что такие классы видимы всем частям системы и нарисовать все связи будет затруднительно. Если придерживаться хорошего стиля, необходимо поместить все основные классы в одну категорию.

Теперь необходимо сделать эту категорию видимой всеми категориями системы. Чтобы не загромождать диаграмму линиями, мы видимую всем категорию пометим словом *global* в левом нижнем углу прямоугольника. Это будет означать, что все объекты, экспортируемые данной категорией, будут видимы для всех категорий, которые импортируют эти объекты.

Пример категории классов. На рис. 5-7 показана диаграмма классов, которая использует категории классов. Это типично организованная диаграмма: абстракции, имеющие физический смысл (именно, *actuators* и *sensors*), расположены на нижнем уровне диаграммы, абстракции по содержанию, близкие к потребностям пользователя, — на верхнем уровне. Категория IPC (*inter process control*) — глобальная категория, поэтому ее ресурсы видимы всем категориям классов этой диаграммы. Мы можем уточнить каждую из шести представленных здесь категорий, для этого необходимо представить соответствующую диаграмму классов, такую, например, как на рис. 5-5.

Шаблоны диаграммы классов

Описанные выше обозначения вполне достаточны для описания структуры классов на высоком уровне абстракции. Просматривая диаграммы, разработчик легко может понять структуру системы. Но одних диаграмм, очевидно, не достаточно, и мы должны обеспечить пользователя более точной информацией. Более конкретно, мы должны иметь средства документального описания каждого класса: его суперклассы, поля и операции.



Рис. 5-6. Обозначения категории классов и видимости.

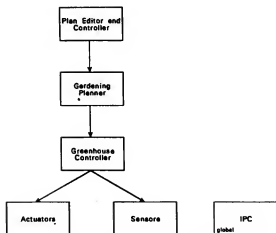


Рис. 5-7. Диаграмма классов тепличного хозяйства на основе гидропоники (высокий уровень абстракции).

Шаблоны класса. Для каждого класса мы имеем шаблон, представленный на рис. 5-8 (шаблон не следует путать с похожим шаблоном языка C++ для параметризованных классов). В шаблоне представлены наиболее важные аспекты класса, описанные в гл. 3. Шаблон подробно определяет класс, но совсем не обязательно заполнять его полностью. На начальном этапе проектирования шаблон заполняется по немногим пунктам и в процессе дальнейшего проектирования дополняется. Если язык, с помощью которого реализуется проект, достаточно выразителен, то можно отказаться от описанного здесь шаблона и описать проект на языке реализации.

Два первых пункта шаблона вполне очевидны. Третий пункт выражает видимость класса в его категории: экспортный, обособленный или импортный. Следующий пункт описывает координатность класса, т.е. сколько экземпляров этого класса можно создать. Обычно координатность равна 0, 1 или n (где n — натуральное число). Место данного класса в иерархической системе определяется следующими двумя пунктами. В зависимости от языка реализации класс может иметь ноль, один и более суперклассов и метаклассов. В любом случае классы, перечисленные в этом пункте, показаны в диаграмме и приводятся в шаблоне всего лишь из соображений полноты и ясности шаблона. Следующий пункт определяет параметры класса, которые могут быть заполнены, только если эти пункты имеют смысл для языка реализации (например, обобщенные параметры для Ada, макросы и параметризованные классы для C++).

Следующие три пункта повторяются в шаблоне 4 раза: три раза для интерфейса и один раз для реализации класса. Если позволяет язык реализации, интерфейс класса можно разбить на три части: доступную, защищенную и обособленную. C++ используют все три части, Object Pascal — только доступную часть. В любом случае это очень важный пункт шаблона, поскольку здесь описывается внешний интерфейс класса. Как описано в гл. 3,

внешний интерфейс класса включает описание экземпляров переменных и переменных класса (описанных в пункте «поле») и операций класса. Для каждого поля мы должны записать его имя, константа или переменная, его класс, диапазон, способ инициализации поля (например, поле L может быть описано как переменная целого типа с диапазоном изменения от 1 до 100). Снова отметим, что нет необходимости заполнять все пункты, можно заполнить пункты, важные для текущего состояния проекта. Шаблон достаточно полон, чтобы выразить все интересные характеристики переменных. Пункт «операции» в действительности содержит список операций, причем каждая операция имеет собственный шаблон, который мы опишем вкратце. Пункт «использование» аналогичен пунктам «суперкласс» и «метакласс» и приводится в этом месте лишь для полноты и ясности описания. Все связи по типу использования показаны графически на объектной диаграмме.

На ранних стадиях проекта разработчик заполняет только доступные, защищенные, обособленные части шаблона. Пункты «поля» и «операции» заполняются после того, как сделаны соответствующие решения о способе реализации проекта.

Шаблон класса содержит четыре поля, которые документируют динамическую семантику, временное и пространственное поведение объектов этого класса. Пункт «конечный автомат» определяет диаграмму переходных состояний; далее мы обсудим подробнее эту диаграмму. Пункт «параллелизм» определяет экземпляры класса как имеющие последовательный, блокирующий или активный тип. Конечно, этот пункт должен быть сопоставлен с параллельностью операций. Например, если класс заявлен как последовательный, его операции не могут быть защищенными, параллельными или многозадачными. Следующий пункт «объем памяти» описывает размер пространства памяти для размещения объектов этого класса. Размер может быть выражен в единицах памяти или в других единицах [6]. Последний пункт шаблона определяет устойчивость объектов класса. Устойчивый объект характеризуется тем, что может существовать после окончания программы, которая его создала. Временный объект существует только во время существования программы.

Суперкласс, метакласс и используемые классы показаны на диаграмме объектов, но они также присутствуют в шаблоне класса. Аналогично некоторые элементы шаблона могут быть показаны в графическом представлении класса. Мы нашли полезным показать координальность, параллельность, устойчивость классов, размещая обозначения этих признаков рядом с обозначением класса (как, например, для обозначения категорий классов мы помещали в нижний левый угол признак «global»). Поэтому разработчик, просматривающий диаграмму объектов, сможет легко уловить важные аспекты проекта, такие, как абстрактность или активность конкретных классов. Еще раз отметим, что у разработчика нет необходимости заполнять все пункты, а лишь те пункты, которые необходимы для документирования важных моментов проекта.

Шаблоны утилиты/класса. Шаблон утилиты/класса показан на рис. 5—9. Поскольку утилита класса представляет собой набор общедоступных подпрограмм (и иногда глобальные константы и переменные), шаблон утилиты можно определить как некоторое подмножество шаблона для классов. Поля и операции, определяемые в интерфейсе утилиты, формируют внешний вид утилиты класса; поля и операции, ответственные за внутреннюю реализацию утилиты, скрываются.

Имя:	идентификатор
Документация:	текст
Видимость:	экспортируемый / обособленный / импортируемый
Множественность:	0 / 1 / n
Иерархия:	
Суперкласс:	список имен классов
Матркласс:	имя класса
Обобщенные параметры:	список параметров
Интерфейс Реализация	
(Общедоступная/Защищенная/	
Обособленная):	
Использование:	список имен классов
Полк:	список полей классов
Операции:	список операций классов
Конечный автомат:	диаграмма перехода состояний
Параллельность:	последовательное выполнение/отсроченное/активное
Объем памяти:	текст
Устойчивость:	статическая / динамическая

Рис. 5-8. Шаблон класса.

Имя:	идентификатор
Документация:	текст
Видимость:	экспортируемый / обособленный / импортируемый
Обобщенные параметры:	список параметров
Интерфейс Реализация	
Использование:	список имен классов
Поля:	список полей
Операции:	список операций

Рис. 5-9. Шаблон для утилиты классов.

Шаблоны операций. Шаблоны класса и утилиты включают список операций. Для нестрогого описания проекта достаточно представить имя операции, ее параметры и семантическое описание операции (в виде текстового описания). Если необходимо более детальное описание проекта, можно воспользоваться шаблоном, представленным на рис. 5-10.

Содержание первых двух пунктов вполне очевидно. Следующий пункт «категория» обеспечивает возможность группирования взаимосвязанных операций (например, как это делается в языке Smalltalk). Например, мы можем сгруппировать операции, изменяющие состояние программы, в группу «модификаторы», а операции, не изменяющие состояние программы, в группу «селекторы». В любом случае категория операций — удобное средство обозначений и особенно полезно в случае, если класс имеет достаточно много операций.

Имя:	идентификатор
Документация:	текст
Категория:	текст
Расширение:	текст
Формат параметров:	список форматов параметров
Результат:	имя класса
Предусловия:	PDL/ диаграмма объектов
Действия:	PDL/ диаграмма объектов
Постусловия:	PDL/ диаграмма объектов
Исключения:	список исключений
Параллельность:	последовательное выполнение/отсроченное/защищенное/параллельное/множественное
Временный ресурс:	текст
Объем памяти:	текст

Рис. 5-10. Шаблон операции.

Пункт «расширение» используется часто на языке реализации CLOS. В этом пункте определяется, является ли операция основным методом или это квалификатор :before, :after, :around. Таким образом, статический обзор операций можно закончить перечислением параметров и результатов (для функции). Динамические свойства операций представлены в следующих четырех пунктах. Мы можем определить семантику функции неформально, описав содержание операций на разговорном языке или же более формально описав

предсостояние, постсостояние и исключительные состояния. Анализ семантики операции может привести к необходимости изучения диаграммы объектов, на которой показаны связи между объектами, участвующими в исходной операции. Также необходимо изучить динамическую семантику операции (показанную на временной диаграмме или с помощью PDL). В любом случае разработчик не должен заполнять все пункты шаблона. Подробность шаблона диктуется текущей потребностью в документации проекта.

Последние три пункта определяют параллельность операции, необходимые временные и пространственные ресурсы операций. В гл. 3 упоминалось, что операция может быть последовательной, т.е. имеется только один канал управления. Альтернативно операция может успешно выполняться в мультипрограммном режиме, но с различными механизмами синхронизации (регулируемой, параллельной, множественной).

Пример шаблона диаграммы классов. На рис. 5-11 показан пример шаблона диаграммы классов для класса Cooler и его операции turnOn.

5.3. ДИАГРАММЫ ПЕРЕХОДА СОСТОЯНИЙ

Состояния и переходные процессы

Диаграмма классов не определяет динамического поведения объектов или классов. Динамическое поведение объектов лучше всего представлено на диаграмме переходов. На этой диаграмме показаны состояния объектов, события, приводящие к переходу из одного состояния в другое, и результат перехода. Диаграмма переходных процессов тесно связана с другими видами обозначений: шаблон класса может включать ссылку на диаграмму переходов, а также действия, описанные в диаграмме переходов, могут ссылаться на другие диаграммы объектов.

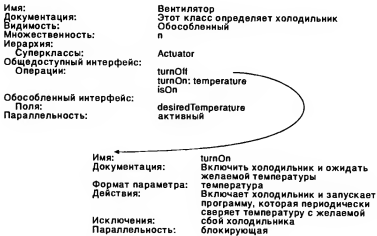


Рис. 5-11. Шаблон диаграммы классов для класса Cooler и его операции turnOn.

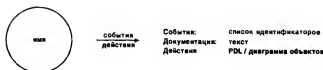


Рис. 5-12. Обозначения для диаграммы переходных состояний.

Состояния. Круг на рис. 5-12 обозначает конкретное состояние. В центре круга — имя состояния; имя должно быть уникальным в пределах диаграммы переходов.

Обычно диаграмма переходов класса не имеет стартовых и финальных состояний. Вновь созданный объект принимает состояние, зависящее от окружения; при уничтожении объекта диаграмма переходов теряет смысл. Имеются два обозначения для стартовых и финальных состояний: стартовое состояние обозначается двойной окружностью, финальное — жирной окружностью.

Переходный процесс. Типом связи между состояниями является переходный процесс, который может приводить к переходу из одного состояния в другое, а также из состояния в такое же состояние. Как показано на рис. 5-12, переходный процесс изображается стрелкой, направленной из исходного состояния в новое. Каждая стрелка может быть помечена именем по крайней мере одного события, инициирующего данный переход, и именем результирующего действия. События и действия не обязательно должны быть уникальными в пределах одной диаграммы, поскольку одно и то же событие может вызвать переход в разные состояния и одно и то же действие может быть выполнено при разных переходных процессах.

Шаблон диаграммы переходов

Шаблон перехода. Так же как для классов, утилит и операций, существует шаблон перехода (рис. 5-12). Поскольку в нашем понимании переход ассоциируется с действием, такой вид диаграммы переходов описывается конечным автоматом Мили, который противоположен конечному автомату Мура, где переходы ассоциированы с состоянием.

Действия, связанные с переходом, могут быть описаны с помощью PDL или же ссылкой на другую диаграмму объектов.

Пример диаграммы переходов. На рис. 5-13 показан пример диаграммы переходов для класса EnvironmentalController тепличного хозяйства на основе гидропоники.

5.4. ДИАГРАММА ОБЪЕКТОВ

Объекты и их взаимосвязь

Диаграмма объектов показывает существующие объекты и их взаимосвязь в логическом проекте системы. Одна диаграмма может представлять часть или всю структуру объектов системы. Обычно для полного документирования проекта необходимо иметь несколько диаграмм. Цель каждой диаграммы объектов — показать семантику ключевых механизмов логического проекта. Классы являются статической частью проекта, а объекты при вы-

полиении программы могут порождаться и уничтожаться. Поэтому мы используем диаграмму объектов, для того чтобы показать динамическую семантику проекта и описать конечные автоматы проекта. Диаграмма объектов представляет собой снимок динамических событий. В этом смысле диаграммы объектов прототипны: каждая из них представляет собой взаимодействия между множеством объектов, причем не имеет значения, какие объекты участвуют во взаимодействии.

Существует важная связь между диаграммами классов и диаграммами объектов (аналогичная связи между классом и его объектом): каждый объект диаграммы объектов представляет собой экземпляр некоторого класса. Кроме того, операции, описанные в диаграмме объектов, должны находиться в соответствии с операциями соответствующего класса. Например, если показано, что объект R посылает сообщение M объекту S, то необходимо, чтобы операция M была определена в классе объекта S. Если мы в процессе разработки проекта исключили операцию N в некотором классе, то надо помнить, что в этом случае, возможно, потребуется исправить некоторые диаграммы объектов.

Для полного документирования логического проекта системы необходимо совместно использовать диаграммы классов и объектов, поскольку они отражают независимые стороны проекта. Диаграмма классов описывает ключевые абстракции системы, диаграмма объектов — важные механизмы, оперирующие этими абстракциями.

Два важных элемента диаграммы объектов — объекты и взаимодействие объектов.

Объекты. На рис. 5-14 показано обозначение для объекта на диаграмме объекта. Оно похоже на обозначение класса, только здесь используем сплошную линию. Имя объекта не обязательно должно быть уникальным и даже может отсутствовать, поскольку объекты в программе могут создаваться динамически без указания имени. Поэтому имя объекта не обязательно должно быть точным и всего лишь должно отражать соответствие объекта какой-либо абстракции, например `aCooler` или `aTemperatureSensor`. Конечно, некоторые объекты могут именоваться точным осмысленным именем, например `GreenHouse7` или `HotWaterValueZone3`.

Свойства объекта можно описывать значками в нижнем левом углу, подобно тому как это делалось для обозначений классов. Наиболее важными свойствами являются параллельность и устойчивость объекта — свойства, описанные в классе объекта. При таком описании объектов одного взгляда на диаграмму объектов достаточно, чтобы определить каналы управления объектами и их устойчивость.

Связь между объектами. Связь между объектами определяется только возможностью одного объекта послать сообщение другому. Поскольку связь такого рода — двунаправленная, для ее обозначения используется ненаправленная прямая линия (рис. 5-14). Мы используем сплошную линию для обозначения связи, которая определена в программной реализации проекта, и пунктирную для внешних по отношению к системе связей (например, для обозначения обратной связи в нагревающих и охлаждающих системах).

На ранних стадиях разработки достаточно документировать взаимосвязь между классами в нестрогих и неполных формах; например, достаточно показать, что объекты могут посылать друг другу сообщения, но не рассматривать спецификации этих сообщений.

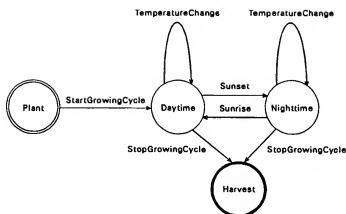


Рис. 5-13. Диаграмма переходных состояний тепличного хозяйства на основе гидропоники.

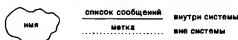


Рис. 5-14. Обозначения для объектов и связей между объектами.

Видимость и синхронизация объектов

Мы можем добавить поясняющие знаки на диаграмме объектов двумя способами.

Видимость объектов. Сначала мы должны уточнить, как объекты видят друг друга. Обозначения, освещающие этот вопрос, не всегда обязательны на диаграмме объектов и присутствуют только в случае необходимости. В гл. 3 были описаны шесть различных форм видимости объектов друг другу. На рис. 5-15 представлены знаки, обозначающие эти формы. Например, если объект R разделяет поле с объектом S, то этот факт на диаграмме объектов обозначается соответствующим знаком, расположенным на линии, соединяющей R с S ближе к объекту R. Видимость объектов друг другу можно показать особым пространственным расположением объектов на диаграмме. Например, разместив активные объекты сверху диаграммы, а пассивные объекты внизу. Аналогично, чтобы показать агрегативность, можно поместить символ одного объекта внутрь символа, обозначающего другой объект.

Синхронизация сообщений. Очень важно показать методы синхронизации взаимодействия объектов. Для этого необходимы новые обозначения. Сообщение, переданное объектом S объекту R, мы обозначаем стрелкой, проведенной вдоль объекта, причем стрелка помечена именем сообщения. Эта стрелка указывает на объект, которому послано сообщение, хотя данные могут передаваться в обратном направлении (например, в направлении линии передаются параметры сообщения, обратно — результат).

S	одна лексическая зона	→	простая
S	одна лексическая зона (общая)	→	синхронная
P	использование параметра	→	отсроченная
P	использование параметра (общее)	→	задержанная
P	использование поля	→	асинхронная
P	использование поля (общее)		

Рис. 5-15. Обозначения для видимости объектов и синхронизации сообщений.

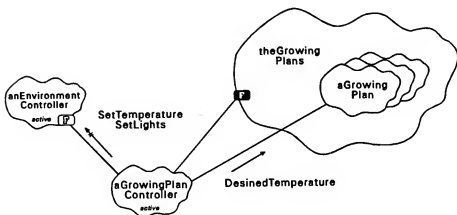


Рис. 5-16. Диаграмма объектов тепличного хозяйства на основе гидропоники.

Для систем с чисто последовательным взаимодействием стрелка является вполне достаточным средством описания взаимодействия объектов. Однако все значительно сложнее, если существует много каналов управления. Например, если два объекта активны одновременно, то сообщение от объекта S к объекту R может быть задержано (возможно, R не готов принять сообщение), или же может привести к фатальной ошибке (S не может долго ждать ответа от R). В системе, возможно, необходимы сообщения, которые прерывают сообщения, посланные другими объектами. В гл. 3 были обсуждены пять типов синхронизации сообщений: простая, синхронная, отсроченная, задержанная и асинхронная. На рис. 5-15 показаны знаки обозначения из работы [7]. Каждый такой знак может быть помечен списком имен сообщений.

Пример диаграммы объектов. На рис. 5-16 показан пример диаграммы объектов. Объект `aGrowingPlanController` посылает два сообщения объекту `anEnviromentController`: `SetTemperature` и `SetLights`. Это сообщение между

активными объектами и является синхронным. Также видно, что `anEnvironmentController` — поле для `aGrowingPlan`, но оно разделяемое, т.е. существуют альтернативные имена для этого объекта.

Объект `theGrowingPlans` обозначает множество, состоящее из индивидуальных объектов `aGrowingPlan`. Чтобы это показать, мы помещаем объекты `aGrowingPlan` внутрь `theGrowingPlans`. Мы используем этот же прием, чтобы показать вложенность классов, утилит и модулей. В проекте, показанном здесь, объект `aGrowingPlanController` может послать сообщение `Desired Temperature` (селектор) любому объекту `aGrowingPlan`. В этом случае семантика передачи сообщения очевидна, поскольку объекты `aGrowingPlan` последовательные.

Шаблон диаграммы объектов

Более полную информацию об объектах и сообщениях можно получить из шаблонов объектов и сообщений.

Шаблон объекта. На рис. 5-17 показан шаблон для объекта. В шаблоне указан класс объекта, поэтому косвенно можно определить операции, которые можно выполнить над объектом. Шаблон обеспечивает информацией об устойчивости объекта, которая в свою очередь должна соответствовать спецификации класса этого объекта. В частности, если в шаблоне класса объекта утверждается, что все экземпляры этого класса временные, то устойчивость объектов может быть статической (объект существует только в течение жизни программы) или динамической (объект создается и уничтожается во время выполнения программы). Если в шаблоне класса утверждается, что экземпляры могут быть устойчивыми, то объекты этого класса могут быть динамическими, статическими или устойчивыми (т.е. объект существует после окончания программы, которая его создала).

Шаблон сообщений. На рис. 5-17 показан шаблон для сообщений. Шаблон описывает операцию, определенную в спецификации класса этого объекта, но в тоже время предоставляет более детальное семантическое описание. Шаблон имеет также временную информацию об операции, например периодичность отправки сообщения и частота посылок. Этот момент важен при проектировании временных ресурсов системы.

5.5. ВРЕМЕННАЯ ДИАГРАММА

События, упорядоченные во времени

Сами по себе диаграммы объектов статичны. Они показывают множество объектов, передающих друг другу сообщения, но не показывают потоки управления и порядок событий. Соответствующая диаграмма переходов также не отвечает на поставленные вопросы, так как только определяет, какие изменения имеют место внутри объекта и не затрагивают взаимодействия между объектами. Таким образом необходимо определить средства документирования динамики отправки сообщений. Мы рассмотрим три типа такой документации. Первый подход очень прост. На диаграмме объектов мы помечаем каждое сообщение числом, равным порядковому номеру отправки этого сообщения. Таким образом, сообщение помечено 1, будет послано первым, затем сообщение 2 и т.д. Такой способ работает хорошо, если порядок отправки строго задан заранее, и становится непригодным в случае условного потока контроля (т.е. если условие *C* верно, то посылается сообщение *M*, иначе сообщение *N*). В этом случае необходимо выбрать второй тип

документации, в котором порядок событий определяется для каждой диаграммы объектов на языке PDL. Язык PDL является выразительным, понятным и поддерживается автоматическими средствами.

Обозначения временной диаграммы

Временные диаграммы третьего типа документирования очень похожи на временные диаграммы, которыми пользуются разработчики аппаратных средств. На рис. 5-18 показано, как мы применяем подобные диаграммы для нашей цели. Видно, что наша *временная диаграмма* — график; по горизонтальной оси расположены метки времени, на вертикальной расположены объекты. Время выражается в абсолютных величинах или в относительных. На вертикальной оси располагаются только те объекты, временная диаграмма взаимодействия которых представляется нам интересной. При движении по графику вдоль оси времени можно определить, какая операция выполняется. Например, начинаем с операции 1 для объекта R. В некоторый момент реализации этой операции вызывает операцию 2 для объекта S, которая в свою очередь посылает сообщение 3 объекту T и т.д. Штриховая линия на диаграмме показывает вложенность сообщений. Например, когда заканчивается операция 3, управление передается обратно операции 2.

Временные диаграммы полезно дополнять символом *, обозначающим момент создания нового объекта, и символом !, обозначающим момент уничтожения объекта. Временная диаграмма может содержать знак создания нового объекта (*) и знак уничтожения объекта (!). Можно также отмечать тупиковые ситуации (недостаток ресурса времени). Например, мы можем проставить на диаграмме необходимый для операции ресурс времени (из шаблона операции). Если в системе существует несколько активных объектов и, следовательно, несколько каналов управления, то необходимо рисовать временную диаграмму для каждого канала управления или же описать каждый канал на языке PDL. Временные диаграммы в этом случае удобнее располагать одна над другой с одинаковыми масштабами времени по оси X.

Обычно диаграмма объектов включает несколько временных диаграмм. Например, на первой временной диаграмме мы можем показать динамическое поведение конкретного механизма, на второй описать асинхронные внешние события, на третьей определить поток управления при исключительных событиях. При этом нет необходимости всегда использовать временные диаграммы для каждой диаграммы объектов, но при документировании критической по времени активности без них трудно обойтись.

5.6. МОДУЛЬНАЯ ДИАГРАММА

Модули и видимость модулей

Диаграммы классов и объектов используются для документирования логических решений проекта системы. Здесь же мы будем рассматривать нотацию для документирования физической реализации проекта, состоящей из программных и аппаратных средств.

Модульная диаграмма определяет распределение классов и объектов в модулях, физически реализующих проект. Одна модульная диаграмма представляет всю или часть модульной архитектуры системы. (Многие авторы пользуются названиями Boochgrams и Gradygram, но мы будем использовать

термин «модульная диаграмма»). Как уже говорилось в гл. 2, некоторые объектно-базовые и объектно-ориентированные языки поддерживают своими средствами концепцию модульности и отличают термин «модуль» от терминов «класс» и «объект». Концепция модульности может быть простой, например в C++ раздельная компиляция файлов, или более сложной, например пакетирование в Ada. Но в любом случае перед разработчиком стоит задача распределения классов, объектов, утилит по отдельным модулям. Языки, которые не поддерживают явно модульную архитектуру, не требуют модульной диаграммы. Два важных элемента модульной архитектуры системы — модули и видимость модуля.

Объект	
Имя:	идентификатор
Документация:	текст
Класс:	имя класса
Устойчивость:	устойчивый / статический / динамический

Сообщение	
Имя:	имя операции
Документация:	текст
Частота:	аperiodическая/периодическая
Синхронизация:	простая/синхронная/отсроченная/задержанная/асинхронная

Рис. 5-17. Шаблон для объектов и сообщений.

Модули. На рис. 5-19 показаны обозначения для различных видов модулей. Эти обозначения имеют достаточно общий вид для эффективного использования их при любых объектно-базированных и объектно-ориентированных языках, но для конкретного языка может быть пригодна только часть этих обозначений. Ada поддерживает все модули, показанные на рис. 5-19, Object Pascal и C++ — только отдельно компилируемые файлы, эквивалентные пакету. Каждый модуль имеет свое имя, которое записывается над обозначением модуля. Имя каждого модуля должно быть уникальным в пределах своей подсистемы. Имя, обведенное прямоугольником, обозначает модуль, экспортируемый подсистемой, а подчеркнутое имя обозначает импортируемый модуль. Вложенность модулей обозначается вложенностью обозначений модулей.



Рис. 5-18. Обозначения временной диаграммы.



Рис. 5-19. Обозначения модулей и видимости модулей.

Обратите внимание на обозначение главной программы. Каждый проект должен иметь по крайней мере один головной модуль, из которой активизируется вся программа. Программный продукт, который выполняется на нескольких компьютерах, связанных сетью, может иметь несколько главных программ.

Видимость модулей. Единственная зависимость между модулями системы — это компиляционная зависимость. На рис. 5-19 компиляционная зависимость обозначается стрелкой. Чтобы показать, что модуль G зависит от модуля H (в терминах C++ включает в себя H), мы проводим стрелку от G к H, тем самым обозначая, что H видим для G. Мы можем дать имя этой стрелки, чтобы более полно документировать существующую связь. Установление таких связей позволяет определить порядок трансляции для Unix make tools и правильно откомпоновать программу.

Пример модульной диаграммы. На рис. 5-20 представлена модульная диаграмма физической реализации тепличного хозяйства на основе гидропоники. На диаграмме представлены шесть модулей: два из них импортированы извне, один экспортирован. Обратите, что обозначение модулей состоит из двух частей: обозначение спецификации пакета (вверху) и обозначение тела пакета. Эта модульная диаграмма представляет достаточно типичную структуру проекта: импортируются ресурсы, необходимые для реализации, и экспортируется пакет, реализующий интерфейс.

Классы Heater, Cooler и Lights помещены в импортируемом пакете GreenhouseActuators. Класс EnvironmentalController и его утилиты находятся в локальном модуле EnvironmentalControllers.

Подсистемы

Как было показано в гл. 2, большие системы можно разложить на сотни модулей. Но такую систему трудно понять. Поэтому разработчики пытаются объединить отдельные модули в какие-то группы. Ниже мы представим обозначения для модульных подсистем, аналогичные обозначениям категорий классов. Подсистема представляет собой набор логически связанных модулей и является полезным орудием при проектировании больших систем.

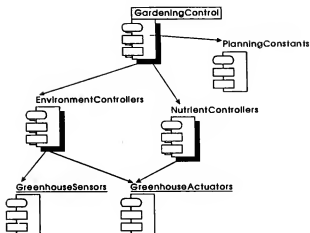


Рис. 5-20. Модульная диаграмма тепличного хозяйства на основе гидропоники.

К сожалению, большинство языков не поддерживает концепцию подсистем; создание подсистем не одно и то же, что создание вложенных модулей. Поэтому мы утверждаем, что язык, даже такой сложный, как Ada, не является достаточным для полной реализации подсистем; пакетирование — необходимое средство декомпозиции, но отнюдь не достаточное. К счастью, многие программные средства позволяют работать с группами модулей. Целью создания подсистем является возможность документирования решений относительно видимости модулей в подсистеме.

Обычно большая система имеет одну высокоуровневую модульную диаграмму, состоящую из подсистем самого высокого уровня абстракции. С помощью такой диаграммы разработчик может понять физическую структуру всей системы. Каждая подсистема является некоторой модульной диаграммой. Модульная диаграмма может содержать или только модули, или только подсистемы.

Обозначение подсистем. Обозначение подсистемы представлено на рис. 5-21. Правила именования подсистем и прикрепленные знаки, такие же, как и для имени модулей. Подсистемы связаны друг с другом компиляционной зависимостью, поэтому обозначение компиляционной зависимости применимо к подсистемам.

Диаграмма подсистем почти аналогична диаграмме классов, представленной на уровне категорий классов. Мы говорим «почти», поскольку есть различия в типах иерархий, характерных для этих двух структур. Категории классов расположены по иерархии типа «разновидность», подсистемы расположены в иерархии типа «составная часть», поскольку подсистемы строятся из нескольких подсистем нижнего уровня.

Пример подсистемы. На рис. 5-22 представлена модульная диаграмма тепличного хозяйства на основе гидропоники. На ней представлены четыре подсистемы. Классы категорий Actuators, Sensors, IPC (на рис. 5-7) пред-

ставлены в подсистеме GardeningSystemDevices. Классы, ассоциированные с категорией GreenhouseController, включены в подсистему GardeningSystemController; классы остальных двух категорий распределены между подсистемами GardeningSystemPlanner и GardeningSystemPlanDatabase.

Шаблон модульной диаграммы

Шаблон статического описания модуля. Графически представленная модульная диаграмма позволяет легко понять модульную структуру проектируемой системы. Для более полной документации еще рассмотрим шаблон модуля, показанный на рис. 5-23.

Наиболее важным пунктом шаблона является список деклараций объектов, содержащихся в этом модуле. Список может включать классы, объекты, утилиты и другие специфичные для языка реализации объекты. Практически оказывается вполне достаточным показать на модульной диаграмме распределение основных классов и объектов, оставив менее значительные объекты без внимания. Напомним, что нотация проекта преднамеренно опускает все незначительные детали.

Возможно, что имя модуля в диаграмме модулей может отличаться от имени файла этого модуля. Например, в CLOS имя пакета совпадает с именем файла, в котором он хранится, но в Object Pascal возможна ситуация, когда пакет UDialogs хранится в файле UDiag.p. Если есть необходимость проследить местонахождение модулей в файлах, то шаблон модуля можно дополнить пунктом, содержащим имя файла.



Рис. 5-21. Обозначение подсистем.

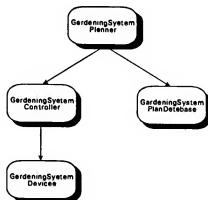


Рис. 5-22. Модульная диаграмма тепличного хозяйства на основе гидропоники.

Имя:	идентификатор
Документация:	текст
Декларация:	список деклараций

Рис. 5-23. Шаблон модуля.

Шаблон динамического описания модуля. Многие рассмотренные выше диаграммы имели статическую и динамическую семантику. Разумно было бы считать, что модульная диаграмма является статическим описанием проекта. Тем не менее в некоторых применениях модули могут иметь и динамическую семантику. Например, в случае малых ресурсов памяти разработчики должны предусмотреть возможность копирования модулей с использованием оверлейной техники программирования. Динамическую семантику модуля мы можем описать с помощью временных диаграмм или средствами PDL.

5.7. ДИАГРАММЫ ПРОЦЕССОВ

Процессоры и приборы

Структура процессов. В гл. 2 мы уже говорили, что очень большая система может потребовать программного обеспечения, состоящего из многих обособленных программ, выполненных на различных компьютерах. Для решения задачи распределения аппаратных ресурсов мы будем пользоваться новым типом диаграмм — диаграммой процессов, которая содержит всю или часть процессорной архитектуры системы. Обычно проект включает одну диаграмму процессов, но для сложной системы их может быть несколько. Диаграмма процессов полезна и в случае однопроцессорной реализации проекта, особенно если в проект вовлечены другие активные приборы и возможно существование параллельных процессов.

Три наиболее важных элемента процессорной архитектуры — процессоры, приборы и соединения. Под процессором мы понимаем часть аппаратного обеспечения, которая может выполнять программу; приборы не обладают такой возможностью.

Процессоры и приборы. На рис. 5-24 показаны обозначения процессоров и приборов. Правила именования процессоров и приборов те же, что и для модулей. Обозначения на рисунке стандартные, но нет причин ограничиваться только такими обозначениями. В любом случае обозначения должны содержать в себе признаки, обозначающие процессор и прибор. Например, мы можем определить обозначения однокристального процессора, дискавода, терминала, АЦП и затем использовать их в диаграмме процессов. Диаграмма процессов, таким образом, будет описывать физические средства проекта.

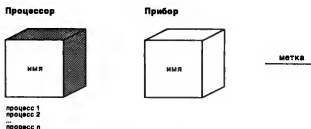


Рис. 5-24. Обозначения процессоров, приборов и соединений.

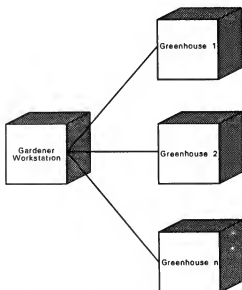


Рис. 5-25. Диаграмма процессов тепличного хозяйства на основе гидропоники.

Под каждым обозначением процессора можно проставлять имена программ и процессов, выполняющихся на этом процессоре, и способы их диспетчеризации. Обозначения диспетчеризации мы вскоре разработаем.

Соединение процессоров и приборов

Соединения. Процессоры и приборы должны взаимодействовать друг с другом. Используя линию, показанную на рис. 5-24, мы можем соединять приборы и процессоры, процессоры и процессоры, приборы и приборы. Обычно соединения представляют собой некоторый кабель (например, линия RS 232, Ethernet сеть, волоконно-оптический кабель), хотя они могут обозначать и не такую прямую связь (например, спутниковую).

Соединения — обычно двунаправленные; для одностороннего соединения линия, его обозначающая, может иметь стрелку. Для полной документации линия, обозначающая связь, должна иметь метку.

Пример диаграммы процессов. На рис. 5-25 представлен пример диаграммы процессов для тепличного хозяйства на основе гидропоники. Надо отметить, что система содержит несколько компьютеров — один центральный компьютер и по одному для оранжерей (GreenHouse). Программы, выполняемые в оранжереях, не могут взаимодействовать между собой непосредственно; такое взаимодействие обеспечивает центральный процессор. Для простоты мы решили не показывать приборы данной системы.

Шаблон диаграммы процессов

Диаграмма процессов позволяет увидеть физическую реализацию проекта. Шаблон диаграммы процессов содержит дополнительную информацию о процессорах, приборах и соединениях.

Процессор		
Имя:	идентификатор	
Документация:	текст	
Характеристики:	текст	
Процессы:	список процессов	
Диспетчеризация:	приоритетный/неприоритетный/через управляющую программу/циклический/ручной	
Процессор		Прибор и соединения
Имя:	идентификатор	Имя: идентификатор
Документация:	текст	Документация: текст
Приоритет:	целое число	Характеристики: текст

Рис. 5-26. Шаблоны для процессоров, процессов, приборов и соединений.

Шаблоны для процессоров и процессов. На рис. 5-26 представлены шаблоны для процессоров и процессов. В шаблон процессора мы можем записать характеристики компьютера (завод-изготовитель, номер, объем памяти). Более важным является пункт, определяющий процесс; выполняемый на данном процессоре процесс может представлять собой головной модуль (описанную в модульной диаграмме) или активный объект (определенный в диаграмме классов или объектов). Шаблон имеет пункт, определяющий канал управления, и его приоритет (если такой имеется). Если система проста, то мы имеем всего лишь один процесс (головной модуль); более сложные системы имеют несколько активных процессов. Возможно, далеко не все из них активны одновременно, тем не менее мы должны предусмотреть самые худшие варианты. Необходимо определить оптимальный график загрузки процессоров, чтобы избежать возможности тупиковой ситуации.

Диспетчеризация процессов. Мы должны некоторым образом определить порядок выполнения процессов на одном процессоре. Ниже приведены пять способов подобной диспетчеризации. Шаблон процессора должен содержать один из этих способов.

Приоритетный	Процесс с более высоким приоритетом выгружает процесс с меньшим приоритетом; процессы с одинаковым приоритетом выполняются процессором в течение небольшого кванта времени, затем загружается следующий процесс, и таким образом все процессы равноправно используют ресурсы процессора
Неприоритетный	Текущий процесс выполняется на процессоре до тех пор, пока он сам не уступит контроль над процессором
Циклический	Контроль управления переходит от одного процесса к другому, каждый процесс имеет для выполнения небольшой промежуток времени — фрейм
Управляющая программа	Конкретный алгоритм управляет загрузкой процессов
Ручной	Процессы запускаются пользователем, например с клавиатуры

Для более полного описания диспетчеризации можно включить в шаблон временную диаграмму или фрагменты PDL. Временная диаграмма и PDL будут полезны для описания динамического поведения программ и объектов, которые могут мигрировать от процессора к процессору.

Шаблон соединений. На рис. 5-26 показан шаблон для соединений. Для программистов шаблон соединений имеет второстепенное значение, хотя для пользователей системы он может быть полезен.

5.8. ПРИМЕНЕНИЕ СИСТЕМЫ ОБОЗНАЧЕНИЙ

Продукты объектно-ориентированного проектирования

В этой главе описаны основные продукты объектно-ориентированного программирования. Обычно проект включает ряд диаграмм классов, объектов, модульных диаграмм, диаграмм процессов. Самый абстрактный уровень описания проекта делится на три части: диаграмма классов проекта с самым высоким уровнем абстракции (определяет основные абстракции логической части проекта), модульная диаграмма (показывает основные моменты программной реализации проекта) и диаграмма процессов (показывает ключевые моменты физической реализации проекта). Основные механизмы взаимодействия показаны на различных диаграммах объектов.

Все диаграммы взаимосвязаны друг с другом, что позволяет проследить связь между реализацией проекта и его спецификацией. На диаграмме процессов может быть обозначена главная программа, которая определена в некоторой модульной диаграмме. Модульная диаграмма в свою очередь содержит описание набора классов и объектов; описания каждого из классов или объектов из этого набора можно найти на соответствующих диаграммах классов и объектов.

Система обозначений, описанная выше, может быть использована при ручной документации проекта; для больших проектов, очевидно, хотелось бы иметь автоматическую поддержку. Автоматическая система помощи могла бы проводить проверку на корректность и полноту документации, помогла бы разработчику легко и быстро просматривать проектную документацию. Глядя на модульную диаграмму, разработчик, возможно, захочет рассмотреть некоторые механизмы взаимодействия — автоматическая система определит и покажет соответствующий объект. Для каждого объекта из диаграммы объектов автоматическая система документации сможет показать спецификацию соответствующего класса этого объекта и также показать место этого класса в иерархической структуре классов. Если объект является активным, автоматическая система поможет определить процессор этого объекта. Автоматическая система избавит разработчика от рутинной работы, от необходимости помнить множество незначительных деталей, тем самым помогая ему сосредоточиться на творческой части работы.

Описанная система обозначений применима как для небольших систем, так и для больших. В гл. 6 и 7 показано, что эта система обозначений очень удобна при последовательном итеративном подходе к процессу проектирования. Нет необходимости считать диаграммы завершенной работой, скорее всего диаграммы эволюционируют в процессе проекта и документируют новые проектные решения.

Мы также нашли нашу систему обозначений, не зависящей от языка реализации. Поэтому она применима для всех объектно-базовых и объектно-ориентированных языков.

До сих пор мы говорили о синтаксисе и семантике графической нотации. В следующих двух главах мы рассмотрим процесс объектно-ориентированного проектирования. Все эти обсуждения будут заканчиваться демонстрацией пяти примеров объектно-ориентированного проектирования.

Заключение

- * Проектирование не заключается в рисовании диаграмм: диаграммы описывают существенные детали проекта.
- * При проектировании сложных систем очень важно иметь возможность смотреть на проект с различных точек зрения (логическая и физическая структура, статическая и динамическая семантика).
- * Система обозначений объектно-ориентированного проекта включает четыре базовые диаграммы (диаграмма классов, диаграмма объектов, модульная диаграмма и диаграмма процессов) и две вспомогательные (диаграмма переходов и временная диаграмма).
- * Диаграмма классов определяет существующие классы и их связь в логическом проекте системы; диаграмма классов представляет все или часть классов проекта.
- * Диаграмма объектов определяет объекты и их взаимодействие в логическом проекте системы; диаграмма объектов представляет все или часть объектов системы и иллюстрирует основные механизмы взаимодействия. Диаграмма объектов представляет собой фотографию событий или конфигурации системы.
- * Модульная диаграмма определяет распределение классов и объектов в модулях программной реализации системы; модульная диаграмма показывает всю или часть модульной архитектуры системы.
- * Диаграмма процессов определяет распределение процессов между процессорами физической реализации системы; диаграмма процессов показывает всю или часть процессорной архитектуры системы.
- * Диаграмма переходов определяет пространство состояний экземпляров конкретного класса, события, приводящие к переходам из одного состояния в другое, и результаты такого перехода.
- * Временная диаграмма определяет динамическое взаимодействие между объектами.

Дополнительная литература

Книга Martin и McClure [H, 1988] — основной справочник по различным системами обозначений.

Ранние версии нотации появились в книге Booch [F, 1981]. В дальнейшем были заимствованы элементы семантических сетей (Stillings et al. [A, 1987], Barг и Feigenbaum [J, 1981]), теории моделей (Ross [F, 1987]) и сетей Petri (Peterson [J, 1977], Saharaoui [F, 1987], Bruon and Balsamo [F, 1986]). Обозначения для объектов и пакетов были частично заимствованы из работ Intel над iARX 432 [D, 1981]. Обозначения для диаграммы объектов взяты из работ Seidewitz [F, 1985]. Обозначения параллельных семантик заимствованы из работ Buhz [F, 1988, 1989]. Диаграммы переходных процессов развиты из работ Harel [F, 1987, 1988].

Другие способы документирования объектно-ориентированного проекта описаны в работах Fischer [C, 1987], Kelly [F, 1986], Grosch [F, 1983], Cunningham и Beck [F, 1986], Kleyn и Gringrich [K, 1988], Schwan и Matthews [K, 1986].

Глава 6

Процесс

Начинающий программист всегда старается найти такое волшебное средство, эпохальное технологическое новшество, которое мгновенно упростит процесс создания программ. Программист-профессионал знает, что подобного средства нет и не может быть. Новичкам хочется следовать известным рецептам; профессионалы же знают, что любой жесткий подход приведет к созданию почти бесполезного продукта. Наконец, когда программист-новичок начинает составлять документацию на свою разработку, он заботится прежде всего о том, как она будет выглядеть для покупателя, а не о том, какую информацию можно в ней найти. Профессионал же понимает всю важность документации, но никогда для него требования к форме продукта не будут определять процесс его создания.

Процесс объектно-ориентированного проектирования — полная противоположность жесткому подходу. Он представляет собой поступательный итеративный процесс, результаты которого оформляются постепенно в процессе разработки.

6.1. ПРОЕКТИРОВАНИЕ КАК ПОСТУПАТЕЛЬНЫЙ ИТЕРАТИВНЫЙ ПРОЦЕСС

Возвратное проектирование

Каким должно быть проектирование: сверху вниз или снизу вверх? Этот вопрос с религиозным рвением часто обсуждался в среде специалистов по программному обеспечению. Нисколько не умаляя знаний программистов, следует признать, что специалисты по аппаратным средствам имеют больший опыт в этой области, поэтому мы задали этот вопрос Дрюку — архитектору компьютерных систем и проектировщику СБИС [1]. Его ответ стал для нас откровением. Предположим, что требуется набрать людей в организацию, занимающуюся разработкой и наладкой достаточно сложных аппаратных средств. Можно использовать горизонтальный набор, когда в увеличивающейся прогрессии нам придется нанимать специалистов по каждой из подсистем (начиная с архитекторов систем), затем специалистов по логическому проектированию, конструкторов схем и т.д. Это пример проектирования сверху вниз, когда мы приглашаем специалистов, которых Дрюк называет «высокими худыми людьми», потому что каждый из них обладает узкими и глубокими знаниями в своей области. Можно пойти по другому пути и нанять проектировщиков — «специалистов на все руки», которые более менее представляют себе весь проект от начала до конца — от архитектурных концепций до отдельных схем. Дрюк называет таких специалистов «короткими толстыми людьми». Нередко, к сожалению, из-за сложности задачи разработки программного обеспечения нам приходится искать специалистов, которых можно назвать «высокими толстыми людьми».

Опыт подсказывает, что проектирование не может осуществляться только сверху вниз или только снизу вверх. Дрюк считает, что хорошо структурированные сложные системы можно создать методом «возвратного проектирования». В этом методе основное внимание уделяется процессу по-

ступательного и итеративного развития путем совершенствования различных, но тем не менее совместных между собой логических и физических моделей системы. Мы считаем, что возвратное проектирование составляет основу процесса объектно-ориентированного проектирования.

Любям, формально обученным методам структурного проектирования, процесс объектно-ориентированного проектирования может показаться слишком произвольным и нечетким. Мы этого не отрицаем, но должны также заметить, что никому еще не удалось организовать творческий процесс, жестко определив все возможные варианты действий. Конечно, чем больше мы знаем о проблеме, которую предстоит решить, тем легче ее решить. Если, например, вы только что возвели один дом своими руками, построить второй вам будет значительно легче, потому что вы знаете, что надо делать и в какой последовательности. Аналогичная ситуация возникает и при разработке программ. Действительно, отчасти это является причиной довольно высокой производительности труда, о которой так любят распространяться некоторые «фабрики» программных средств [2]. В таких организациях решаемая задача обычно хорошо определена (например, обработка платежных ведомостей), и под нее уже созданы механизмы решения. В этих областях процесс проектирования почти полностью систематизирован и разработчики новой системы уже знают наиболее важные абстракции, механизмы, которые необходимо применить, и в основном представляют себе поведение будущей системы. Подобная ситуация предполагает возможность повторного использования не только отдельных программных компонент, но и самого метода вместе с требованиями к системе. Элемент творчества имеет важное значение и в такого рода разработках, но он находит здесь ограниченное применение, так как основные проблемы проектирования системы уже решены. Производительность в результате оказывается естественно высокой по сравнению с уникальными разработками.

Эксперименты Картиса и его коллег [3] подтверждают эти наблюдения. Картис изучал работу профессиональных разработчиков программного обеспечения, фиксируя видеокамерой их действия и затем анализируя их содержание (анализ, проектирование, ввод данных и т.д.) и время, затрачиваемое на их выполнение. В результате этих исследований он сделал вывод о том, что «создание программ представляет собой, по-видимому, ряд взаимопересекающихся итеративных и беспорядочных процессов, проходящих под непосредственным контролем... Сбалансированный процесс разработки сверху вниз оказывается, по-видимому, особым случаем, соответствующим абсолютно правильно выбранной схеме решения или простой задаче... Хорошие разработчики одновременно могут охватить многие уровни абстракции и работать в них».

Как уже говорилось в гл. 1, большинство программных систем достаточно уникально и их разработчики имеют, следовательно, весьма ограниченный опыт. В таких условиях полезно в процессе разработки периодически анализировать полученные результаты, совершенствовать промежуточный продукт в соответствии с новыми представлениями и повторять этот процесс, пока проект не будет корректным образом полностью реализован. Хайляйн предлагает в том случае, «когда вы имеете дело с непонятной задачей, решить ту ее часть, которая вам понятна, а потом вернуться к проблеме снова» [4]. Это еще одно определение понятия «возвратное проектирование».

Объектно-ориентированное проектирование: действия и результаты

Процесс объектно-ориентированного проектирования является возвратным процессом; модели проектирования, рассмотренные в гл.5, являются продуктами данного процесса. Объединение понятий объектно-ориентированного и возвратного проектирования дает нам большие потенциальные преимущества. Богатая система нотаций позволяет, например, моделировать задачу несколькими возможными путями и фокусировать внимание в разное время на относительно независимых частях проблемы. Эволюционный характер процесса проектирования дает возможность использовать ту модель, которая в настоящий момент наиболее нужна для решения поставленной задачи. Мы могли бы, например, сначала составить схему структуры классов системы с помощью диаграмм классов, затем разработать некоторые механизмы, используя эти абстракции, и задокументировать их в серии объектных диаграмм. Проектируя детали этих механизмов, мы возвращаемся обратно к структуре классов, устанавливаем протоколы для каждого класса и, возможно, реорганизуем порядок наследования, чтобы лучше использовать принятую общность выразительных средств. По окончании процесса проектирования мы получаем ряд диаграмм, соответствующих различным, но при этом четким и согласующимся моделям системы, появившимся в результате эволюции предыдущих устойчивых, хотя и не столь подробных моделей.

Ступенчатый, итеративный процесс объектно-ориентированного проектирования отличается от традиционного процесса разработки программного обеспечения, где приходится мириться с лавинообразным нарастанием сложности в процессе создания программ. Если говорить откровенно, надо признать, что традиционный подход имеет ряд неустраняемых недостатков и нарушает многие инженерные принципы. И дело не только в том, что процесс проектирования ни в коем случае не должен быть жестко структурирован. Кроме этого, он обязательно должен предполагать эволюционность развития. Подобный подход соответствует спиральной модели развития программного обеспечения, которая предложена Бемом и «выдвигает на первый план подход к развитию программного обеспечения, ориентированный прежде всего на риск, а не на прототипы и спецификации» [5].

Объектно-ориентированное проектирование это не тот процесс, который начинается с определения требований, завершается составлением схемы применения системы, а между этими двумя событиями представляет собой нечто непонятное и таинственное. В соответствии с нашими представлениями процессу объектно-ориентированного проектирования соответствует примерно следующий порядок событий:

- * Идентификация классов и объектов данного уровня абстракции.
- * Идентификация семантики классов и объектов.
- * Идентификация связей между классами и объектами.
- * Использование классов и объектов.

Это постепенный процесс: идентификация новых классов и объектов обычно заставляет нас модифицировать семантику и связи между существующими классами и объектами. В то же время это итеративный процесс: использование классов и объектов часто приводит к необходимости разработки и создания новых классов и объектов, присутствие которых упрощает процесс проектирования.

С чего начать процесс объектно-ориентированного проектирования и чем его завершить? Начинать надо с создания классов и объектов, формирующих словарь проблемной области. Конечной целью является «идентификация всех объектов, принадлежащих проблемной области и играющих заметную роль, определение связей между объектами и наделяние каждого объекта определенными функциями, с помощью которых он будет совершать действия над другими объектами и другие объекты будут совершать действия над ним, а также анализ этих функций на том уровне, который больше всего подходит для их понимания» [6]. Таким образом, мы можем остановить процесс проектирования, если увидим, что больше нет ключевых абстракций и механизмов, не включенных в нашу модель, и используя уже существующие компоненты программного обеспечения, применяя их к созданным классам и объектам, мы можем обеспечить требуемое поведение нашей системы. В этом случае не имеет смысла проектировать что-то новое; главной задачей становится применение созданных нами моделей.

Ниже мы подробнее рассмотрим каждый из четырех шагов процесса проектирования и их результаты. В гл. 7 мы приведем некоторые примеры таких процессов и проанализируем их в основном с точки зрения руководителей, обязанных управлять ходом работ и использовать объектно-ориентированное проектирование, так как этот процесс требует знания ряда основных положений и предполагает совершенно другой подход к распределению ресурсов и к управлению, чем для традиционных процессов. В гл. 8-12 рассмотрен широкий спектр различных объектных и объектно-ориентированных языков программирования.

6.2. ИДЕНТИФИКАЦИЯ КЛАССОВ И ОБЪЕКТОВ

Действия

Первый шаг состоит из двух действий: создание основных абстракций, принадлежащих проблемной области (важнейших классов и объектов), и разработка основных механизмов, обеспечивающих требуемое поведение объектов, которые в свою очередь обеспечивают функционирование системы. Каким образом мы выделяем эти абстракции и механизмы? Анализируя конкретную предметную область с помощью методов, описанных в гл. 4, разработчик при этом должен действовать по существу как абстракционист. Изучая требования к задаче и (или) обсуждая их в процессе дискуссий со специалистами по данной проблеме, разработчику необходимо освоить терминологию проблемной области, основные теоретические положения, их роль. Затем приблизительно сформировать классы и объекты вплоть до самого высокого уровня абстракции.

Следует отметить, что пока это лишь классы и объекты «в начальном приближении», ведь на данном этапе разработки мы только приступаем к определению границ наших абстракций. Мы предполагаем, что эти границы будут видоизменяться с течением времени в процессе того, как мы начнем обнаруживать что-то общее у наших классов и объектов и модифицировать введенные нами механизмы взаимодействия. Хотя их внешний вид может слегка изменяться, обычно оказывается, что и классы, и объекты, определенные на ранних стадиях проектирования, сохраняются на протяжении всего процесса создания системы. Этого можно было ожидать, так как соответствие продуктов объектно-ориентированного проектирования реальной физической модели является одним из принципов создания объектной модели.

Рассмотрим проект гидропонно-садовой системы. Даже поверхностное ознакомление с требованиями к системе позволит определить ее основные компоненты: растущие растения, сельскохозяйственные культуры и окружающие их параметры среды (например, температура воздуха и концентрация питательных веществ). Данные абстракции существенны для рассматриваемой предметной области и, таким образом, входят в первоначальный список классов и объектов проекта решения нашей задачи.

Результаты

Результаты первого шага могут быть различными: иногда достаточно лишь составить список имен важнейших классов и объектов, так чтобы эти имена несли определенную смысловую нагрузку в соответствии с местом и функциями того или иного объекта или класса в вашей системе. Мы называем это «списком имен», который всегда открыт для продолжения [7]. Одни элементы данного списка могут оказаться классами, другие — объектами, а третьи — свойствами объектов. Результатом первого шага может стать не просто определение имен, но также заполнение шаблонов соответствующих классов и объектов, т.е. формальное определение смысла выделенных абстракций и механизмов.

В большинстве случаев этот шаг занимает немного времени по сравнению с остальными тремя шагами. Нередко руководитель проекта в одиночку составляет предварительный список классов и объектов, а затем обсуждает этот список с коллегами, проводя своего рода проверку умственных способностей подчиненных. Основная цель составления такого списка заключается в выработке терминологии общения между разработчиками. Проектировщику также может оказаться полезно начертить кое-какие диаграммы классов или объектов, чтобы уяснить механизмы совместного их функционирования. Если вдруг важное значение приобретает проблема физической декомпозиции системы, параллельно можно начертить модульные диаграммы. Если же в системе присутствуют элементы аппаратного обеспечения, полезно составить необходимые диаграммы процессов.

6.3. ИДЕНТИФИКАЦИЯ СЕМАНТИКИ КЛАССОВ И ОБЪЕКТОВ

Действия

Второй шаг — основной — заключается в определении содержания классов и объектов, заданных на предыдущем шаге. Здесь разработчик действует как бы со стороны, анализируя каждый класс с точки зрения перспективы его взаимодействия с другими классами и объектами и определяя механизмы взаимодействия классов и объектов.

Этот шаг гораздо труднее первого и занимает больше времени: не обойдется без жарких дебатов, выкриков, зубовного скрежета и упоминаний личных имен на совещаниях. Классы и объекты определить легко, договориться о содержании протокола для каждого объекта гораздо сложнее. По этой причине с определенного момента процесс объектно-ориентированного проектирования становится итеративным. Составление протокола для данного объекта может потребовать принятия решений, затрагивающих содержание другого объекта. При этом само существование наших абстракций не ставится под вопрос, происходит лишь смещение границ между ними. Для того чтобы направить данную деятельность в нужное русло, полезно составить

для каждого объекта свой текст, который определял бы цикл его функционирования — от создания до удаления и особенности его поведения.

Проектируя, например, гидропонно-садовую систему тепличного хозяйства, нам необходимо сначала создать класс, экземпляры которого представляют собой выращиваемые растения. Можно так спроектировать этот класс, что его объекты будут содержать знания о всех подробностях процесса выращивания данной сельскохозяйственной культуры, включая знания о том, как составить план такого выращивания. Позднее нам может показаться, что протокол этого класса слишком перегружен, и мы тогда создадим другой класс — менеджер планов выращивания, который знает, как обеспечивать выполнение планов, являющихся частью состояния объектов. Подобные эволюционистские решения приводят к созданию новых классов и меняют внешний вид первоначальных.

Результаты

Результаты данного шага отражают ступенчатый характер процесса объектно-ориентированного проектирования. Выработав конструкторские решения, касающиеся содержания каждого класса и объекта, мы в основном завершим доводку шаблонов, созданных на первом шаге. Для этого необходимо наилучшим образом зафиксировать все статические и динамические свойства каждой основной абстракции и механизма. Мы могли бы также начертить новые диаграммы объектов, отражающие те механизмы, которые были введены на данном этапе процесса. В заключение, мы могли бы создать прототипы отдельных частей проекта с целью анализа его текущего состояния и оценки различных подходов к решению тех подзадач, которые, по нашему мнению, находятся в зоне повышенного риска.

6.4. ИДЕНТИФИКАЦИЯ СВЯЗЕЙ МЕЖДУ КЛАССАМИ И ОБЪЕКТАМИ

Действия

Третий шаг можно рассматривать как продолжение предыдущего. Здесь мы точнее определяем механизмы взаимодействий внутри системы. Что касается основных абстракций, то мы должны определить типы отношений между ними: использование, наследование и т.д. Необходимо также определить статические и динамические свойства объектов и их механизмов.

Существуют две основные причины, заставляющие нас заниматься совершенствованием результатов нашего проекта. Во-первых, мы должны разработать структуры: структуры классов, что приводит к необходимости реорганизации и упрощения системы классов, и структуры совместных организаций объектов, что ведет к обобщению механизмов взаимодействий, включенных в проект. На данном шаге проектирования от разработчика требуется мобилизация всех творческих сил; по степени успеха, достигнутого на данном шаге, ваш проект будет затем оцениваться как просто хороший или как чрезвычайно удачный.

Во-вторых, мы должны определить границы видимости: какие классы и объекты могут видеть друг друга, и, что не менее важно, какие классы и объекты не могут видеть друг друга. Эти решения помогут нам затем предпринять верные шаги при проектировании архитектуры модулей нашей системы. Разработка структур и принятие решений о степени видимости могут также заставить нас провести усовершенствование протоколов для классов,

выделенных на более ранних этапах. В конце концов мы получим общие абстракции и механизмы, определенные только в одном месте.

Метод, который, по-видимому, может помочь при подобных операциях, заключается в использовании карт КФС (класс, функции, связи), предложенных Беком и Каинингемом [8]. Карта КФС — обычная карточка (по одной на каждый класс), на которой написано имя класса, функции, которыми он наделен, и имена классов, с которыми он совместно функционирует. Проектировщик может затем располагать эти карты на столе по своему усмотрению и редактировать их содержание в соответствии с описанием циклов функционирования объектов, составленных на предыдущем этапе.

Рассмотрим еще раз проект гидропонно-садовой системы. Проанализировав уже разработанные механизмы, мы сможем заметить, что механизм записи информации об индивидуальных планах выращивания каждого растения можно применить для записи информации о пользователях редактора этих планов. Такая структура абстракций и механизмов может быть разработана только специалистом, использующим метод возвратного проектирования. Его применение предусматривает возможность создания одного, а не двух механизмов для поддержки сходного функционирования двух разных, не связанных между собой, частей системы.

Следует отметить, что на данном шаге проекта мы все еще рассматриваем существующие абстракции и механизмы как бы со стороны. На первых трех шагах не стоит пытаться рассматривать их как часть нашей системы, потому что мы еще до конца не решили, как та или иная абстракция или тот или иной механизм конкретно будет использован.

Результаты

Результатом данного шага является завершение создания большинства логических моделей проекта. Происходит доводка диаграмм классов, разработанных на предыдущих шагах, с учетом принятых решений о структурах классов и объектов и об их взаимной видимости. Мы окончательно заполняем диаграммы объектов, фиксирующие основные механизмы, которые присутствуют в нашей системе. Составление таких диаграмм на практике — дело довольно простое. Используя нотации из гл. 5, мы сначала рисуем объекты, которые, как мы знаем, определенным образом взаимодействуют друг с другом. Затем мы выбираем пару объектов и определяем, взаимодействуют ли они друг с другом. Если ответ положителен, мы можем задать следующие два вопроса: каким образом эти объекты связаны, и какие сообщения передают они друг другу? Обычно после этого нам приходится изменять протоколы соответствующих классов, а эти изменения в свою очередь могут заставить нас создать некоторые дополнительные структуры. Именно поэтому мы и говорим, что объектно-ориентированное проектирование — итеративный процесс.

С этого момента мы можем начинать создавать необходимые модульные диаграммы нашего решения (или редактировать старые), принимая во внимание те решения о взаимной видимости классов, что мы приняли ранее. Мы стараемся построить классы, объекты и модули, которым было бы проще взаимодействовать друг с другом. Теперь наш продукт находится на такой стадии разработки, когда мы можем оценить его с помощью тех критериев, о которых шла речь в гл. 3. После соответствующего анализа нам, возможно, придется изменить наш проект, чтобы улучшить свойства наших основных абстракций и механизмов.

На данном шаге мы также продолжаем разрабатывать прототипы. Мы можем создавать новые прототипы, чтобы еще раз сравнить различные подходы к проектированию тех элементов системы, которые, на наш взгляд, являются источником повышенного риска. Мы можем также редактировать старые прототипы с целью создания прототипов, обладающих большими функциональными возможностями, которые в конце концов станут частью конечной реализации нашей системы. Действуя по такой схеме, мы всегда будем иметь работающий вариант нашей системы.

6.5. РЕАЛИЗАЦИЯ КЛАССОВ И ОБЪЕКТОВ

Действия

Четвертый шаг — не всегда последний. Предстоит решить еще две основные задачи: принять решения относительно включения в нашу систему конкретных классов и объектов и распределить классы и объекты по отдельным модулям, а программы — по процессорам. На этом шаге проектирования мы впервые рассматриваем каждый объект и класс изнутри, чтобы отметить способы реализации его свойств.

Но даже если наши абстракции и механизмы довольно просты, в этом месте обычно приходится возвращаться к первому шагу и снова проходить весь путь, рассматривая изнутри существующие классы и отдельные модули. Таким образом мы повторяем процесс проектирования, фокусируя внимание на более низких уровнях абстракции. Означает ли это, что объектно-ориентированное проектирование является примером проектирования сверху вниз? Не совсем. На практике мы сначала проектируем более высокие уровни абстракции и включаем в них те классы и объекты, которые явно соответствуют терминологии предметной области. Однако, находясь на любом шаге разработки, нам часто приходится обращаться к более низким уровням абстракций и механизмов, так как они являются теми первичными классами и объектами, на основе которых построены все классы и объекты более высоких уровней. Таким образом, мы опять видим, что объектно-ориентированное проектирование включает процесс возвратного проектирования.

Результаты

На данном шаге мы организуем конкретное взаимодействие между каждым классом и объектом, который играет какую-то роль на рассматриваемом уровне абстракции. Результатом, таким образом, является конечная доработка структуры классов нашей системы, и, в частности, завершение процесса создания шаблонов для каждого класса. Такого же уровня полноты должны достигнуть модульные диаграммы и, если этого требует архитектура системы, диаграммы процессов.

Заключение

- * Процесс объектно-ориентированного проектирования нельзя определить ни как проектирование сверху вниз, ни как проектирование снизу вверх; его можно скорее назвать «возвратным проектированием», что подразумевает ступенчатый и итеративный процесс разработки системы с постепенной модификацией различных, но тем не менее согласованных между собой логических и физических представлений о системе в целом.
- * Процесс объектно-ориентированного проектирования начинается с разработки классов и объектов, формирующих словарь предметной области; он за-

вершается, когда оказываются исчерпаны все базовые абстракции и механизмы или когда разработанные нами классы и объекты могут быть составлены из уже существующих универсальных программных компонент.

- * Первый шаг процесса объектно-ориентированного проектирования включает идентификацию классов и объектов данного уровня абстракции; наиболее важной является разработка основных абстракций и механизмов.
- * Второй шаг заключается в определении свойств этих классов и объектов; здесь разработчику важно действовать как бы со стороны, рассматривая каждый класс с точки зрения взаимодействия его с другими классами.
- * Третий шаг заключается в определении связей между различными классами и объектами; здесь мы устанавливаем, каким образом происходят взаимодействия внутри системы, обращая при этом внимание на статические и динамические свойства основных абстракций и механизмов.
- * Четвертый шаг представляет собой процесс создания классов и объектов; наиболее важным является правильный выбор места в логической структуре системы для каждого класса и объекта, распределение классов и объектов по отдельным модулям, программам и процессам; этот шаг не обязательно является последним, так как для его завершения обычно требуется снова пройти весь процесс от начала до конца, на сей раз обращая основное внимание на более низкие уровни абстракции.

Дополнительная литература

Существует несколько вариантов организации процесса объектно-ориентированного проектирования. Сравнение наиболее популярных подходов можно найти в работах Boehm-Davis and Ross [H, 1984], Kelly [F, 1986], Mannino [F, 1987]. Обзор ряда других методов, включающих объектно-ориентированное проектирование, можно найти у Webster [F, 1988]. Первоначальный вариант процесса, рассмотренного в этой главе, впервые описан Booch [F, 1982]. Berard позднее усовершенствовал данную методологию [F, 1986]. Ряд аналогичных подходов рассмотрен в работах GOOD (General Object-Oriented Design) Seidewitz and Stark [F, 1985, 1986, 1987]; SOOD (Structured Object-Oriented Design) Lockheed [C, 1988]; MOOD (Multiple view Object-Oriented Design) Keth [F, 1988] и HOOD (Hierarchical Object-Oriented Design) CISI Ingenierie and Matra, for the European Space Station [F, 1987]. В работах Wasserman, Pircher и Muller [F, 1988], Mills [H, 1986] и Constantine [F, 1989] рассматривается смешанный подход, включающий объектно-ориентированное проектирование и структурный подход и названный, как и следовало ожидать, объектно-ориентированным структурным проектированием (OOSD). Ряд других ученых предлагают похожие модели проектирования, и здесь мы можем привести достаточно большой список работ: Alabios [F, 1988], Boyd [F, 1987], Buhr [F, 1984], Cherry [F, 1987, 1990], Felsing [F, 1987], Fivesmith [F, 1986], Hines и Unger [G, 1986], Jacobson [F, 1985], Jamsa [F, 1984], Kadie [F, 1986], Masiero и Gevmano [F, 1987], Nielsen [F, 1988], Nies [F, 1986], Rajlich и Silva [F, 1987], и Shumate [F, 1987]. Основные шаги объектно-ориентированного проектирования, рассмотренные в этой главе, аналогичны предложенным в работах Bailin [B, 1988], Barry, Thomas, Altolf и Wilson [C, 1987], Chen [F, 1976], Coad [B, 1989], Date [E, 1986], Goldbend и Kay [G, 1977], Gouda, Han, Jensen, Johnson, и Kain [F, 1977], Henderson [J, 1986], Jackson [H, 1983], Keene [G, 1989], Mascot [H, 1987], Pinson и Weiner [G, 1988], Smith и Smith [E, 1980] и Yourdon [H, 1989].

Глава 7

Традиционные методы

Разработка программного обеспечения остается до сих пор трудоемким процессом; в некотором смысле ее можно назвать кустарным промыслом [1]. Даже в Японии индустрия программного обеспечения «базируется в основном на ручных методах, используемых даже на заключительных этапах разработки» [2].

Опыт показывает, что проектирование не является точной наукой. Рассмотрим, например, разработку сложной базы данных с использованием модели отношений между объектами — одной из основ объектно-ориентированного проектирования. «Если говорить честно, важность отдельных объектов процесса определяется в значительной степени личными вкусами разработчика. В результате этого процесс проектирования не является детерминированным: разные авторы могут составить разные модели одного и того же процесса» [3].

Поэтому мы приходим к выводу о том, что каким бы сложным ни был метод проектирования, какими бы солидными ни были его теоретические основы, нельзя игнорировать практические аспекты проектирования, которые сложились на сегодняшний день для решения практических задач. Это означает, что мы должны принимать во внимание такие практические приемы управления процессом разработки программного обеспечения, как распределение ресурсов, установление промежуточных этапов, управление конфигурацией и проверка версий. Для профессионального разработчика программного продукта это реальность, с которыми надо считаться, если хочешь создать сложную программную систему. Поэтому в данной главе рассматривается применение традиционных методов в объектно-ориентированном проектировании, оценивается их значение в жизненном цикле разработки и влияние на практику управления процессом разработки.

7.1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ В ЖИЗНЕННОМ ЦИКЛЕ РАЗРАБОТКИ

Жизненный цикл разработки программного обеспечения

На рис. 7-1 показаны этапы традиционного цикла разработки программного обеспечения, характеризующегося лавнообразным нарастанием сложности. Во многих компаниях такая схема обычно соответствует этапам процесса разработки систем и ее часто рассматривают как неизбежную. При этом считается, что работа, как вода, перетекает вниз с одного этапа на другой. Часто даже организационная структура таких фирм соответствует приведенной схеме. Но несмотря на силу традиций, недостатки лавнообразной модели признаются почти всеми. Бозм, например, формулирует эти недостатки следующим образом:

- * «Непригодность для разработки сложных программных систем, состоящих из большого числа автономных модулей, а также для организации процесса внесения в систему последующих изменений.
- * Обязательно последовательное выполнение всех этапов разработки.

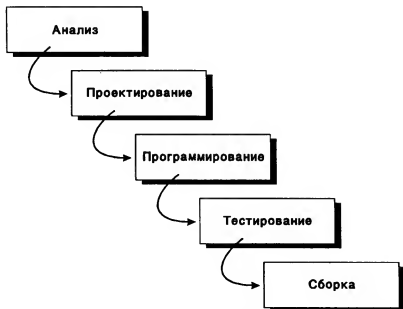


Рис. 7-1. Традиционный лавинообразный цикл разработки программного обеспечения.

- * Несовместимость с эволюционным подходом, который широко внедряется в настоящее время благодаря возможностям быстрого прототипирования и применения алгоритмических языков четвертого поколения.
- * Несовместимость с перспективными методами разработки: возможностям автоматического программирования, трансформации программ и применения вспомогательных средств, основанных на базах знаний» [4].

Следует особо отметить третий недостаток, поскольку, как мы выше не раз говорили, объектно-ориентированное проектирование представляет собой поступательный итеративный процесс, что совершенно не согласуется с одним из основных постулатов сторонников лавинообразной модели, по которому в отлаженный один раз программный продукт очень сложно вносить затем какие-либо изменения. Как пишут Хэтли и Пирбхам: «Это искажает естественный ход процесса работы над системой, который всегда является итеративным и в котором результаты одного из этапов могут изменить решения, принятые на предыдущих этапах» [5]. В первую очередь по этой причине Бозм выдвинул спиральную модель создания программного обеспечения.

Однако, понимая, что объектно-ориентированное проектирование есть последовательный итеративный процесс, не следует отказываться от полезных практических советов по управлению процессом разработки, основанных на опыте применения лавинообразной модели. Важным остается этап анализа, но необходим и четко выраженный этап проектирования. Од-

нако для руководителя и разработчика, привыкших к лавинообразной структуре процесса создания программных систем, объектно-ориентированное проектирование выглядит чуждым и путающим, поскольку некоторые традиционные приемы организации работы теряют в нем всякий смысл. Интеграция, например, в объектно-ориентированном проектировании не выделяется в отдельный этап, а рассматривается как непрерывный процесс.

На рис. 7-2 показана диаграмма жизненного цикла разработки программного обеспечения при объектно-ориентированном проектировании. Мы видим, что проектирование не является отдельным монолитным этапом; скорее оно представляет собой один из шагов на пути последовательной итеративной разработки системы, при этом последовательность шагов может иметь произвольный характер. В следующих разделах мы рассмотрим, каким образом при использовании объектно-ориентированного подхода можно оценить численность разработчиков, определить номенклатуру необходимых промежуточных документов, организовать процессы выпуска программ, обеспечения качества и программной поддержки. Вначале рассмотрим этапы жизненного цикла объектно-ориентированной разработки программного обеспечения.

Анализ

Действия при анализе. На этапе анализа происходят первые встречи разработчиков с будущими пользователями системы, которые, исходя из особенностей поставленной задачи, пытаются найти между собой общий язык. Как считают Меллор и другие: «Целью анализа является описание задачи». Описание должно быть полным, последовательным, доступным для чтения и обзором различными заинтересованными сторонами, позволяющим проводить сравнение с реальными условиями» [6]. Продукт анализа часто используется затем для описания основных функций системы. Говоря о функции, мы не имеем в виду алгоритмический термин. В контексте анализа требований к системе функция — это обособленный, наблюдаемый и контролируемый фрагмент поведения. Например, функцией системы организации информации является обеспечение различных видов поиска в базе данных.

Граница между анализом и проектированием весьма расплывчата. Определение ключевых абстракций предметной области может рассматриваться, например, и как часть анализа, и как часть проектирования. Тем не менее цели анализа и проектирования совершенно различны. При анализе мы пытаемся моделировать окружающий мир, идентифицируя классы и объекты, образующие словарь предметной области; при объектно-ориентированном проектировании мы придумываем абстракции и механизмы, обеспечивающие поведение, которое требует эта модель. Другими словами, анализ определяет требуемое поведение системы, которую мы должны создать, в то время как при проектировании разрабатываются чертежи этой системы.

Необходимо разделять действия, выполняемые при анализе и проектировании. Нет ничего хуже, чем получить документ с требованиями, часть из которых в действительности не является требованиями, а содержит указания о том, что вы должны проектировать систему определенным образом. С другой стороны, опасно пытаться «полиостью» проанализировать систему, даже не подумав заранее о том, как приступить к проектированию. Такой подход не позволил бы успешно завершить анализ.

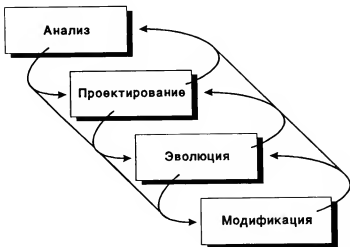


Рис. 7-2. Цикл разработки программного обеспечения с использованием объектно-ориентированного подхода.

Методы анализа. Существует много методов анализа, которые можно использовать до начала объектно-ориентированного проектирования. Наиболее известным методом является структурный анализ, изложенный в работах Йордона [7], Де Марко [8], Гейна и Сарсона [9], Уорда и Меллора [10], Хетли и Пирбхай [11]. Мы считаем, что результат любого из многочисленных методов структурного анализа можно использовать как исходный материал для объектно-ориентированного проектирования [12]. Как мы уже говорили в гл. 4, анализ абстракций связывает диаграммы потоков данных, полученные при структурном анализе, с классами и объектами объектно-ориентированного проектирования.

Заманчиво предвзирать объектно-ориентированное проектирование структурным анализом, поскольку этот метод хорошо известен, им владеют многие специалисты, поддержка его обеспечивается обширными библиотеками программ. Однако структурный анализ не рекомендуется использовать на начальном этапе объектно-ориентированного проектирования, так как при анализе диаграмм потоков данных трудно отделить составную модель задачи от проектного решения. Кроме того, на результаты структурного анализа иногда влияет алгоритмически-ориентированный подход к задаче, что сильно усложняет объектно-ориентированное проектирование.

Современной тенденцией является применение объектно-ориентированного анализа, описанного в работах Шлеера и Меллора [13], Коуда и Йордона [14]. По определению Смита и Токи, объектно-ориентированный анализ включает «процесс идентификации и моделирования основных классов и объектов, логических отношений и взаимодействий между ними» [15]. Этот процесс достаточно близок к объектно-ориентированному проектированию. Практически это означает, что результаты объектно-ориентированного анализа могут непосредственно использоваться на начальном этапе процесса объ-

ектно-ориентированного проектирования. Далее разработчик лишь модифицирует результаты анализа, вводя новые абстракции и механизмы, которые позволяют более эффективно использовать уже идентифицированные классы и объекты.

Перед началом объектно-ориентированного проектирования могут применяться и другие методы анализа, включая SADT [16], SREM [17], моделирование соотношений между объектами. Вне зависимости от примененного метода важно, что результат анализа предоставляет разработчику достаточно полную модель задачи, к решению которой он может приступить.

Проектирование

Начало процесса проектирования. Когда начинается проектирование? Практически тогда, когда мы имеем некую (возможно полную) формальную или неформальную модель поставленной задачи. Если мы начнем проектирование слишком рано, исходных сведений о задаче может не хватить для того, чтобы принимать обоснованные компромиссные решения при проектировании. Если проектирование начинается слишком поздно, мы рискуем потратить много сил на подробный анализ, который «обрушится» на разработчика лавиной лишних и ненужных подробностей. Поэтому мы предлагаем следующую стратегию разработки, которую назовем «немного анализа, немного проектирования». В этой стратегии каждый шаг проектирования направляет процесс анализа на выявление тех аспектов системы, которые важны для получения решения.

В реальных условиях специалисты, занимающиеся анализом и разработкой, не всегда могут быть тесно связаны друг с другом. Так бывает, когда разработку частей программного обеспечения выполняют субподрядчики. Однако это не должно препятствовать применению стратегии «немного анализа, немного проектирования». Попробуйте проектировать, базирясь на вашем понимании требований в данный момент времени. Остановитесь, изучите достоинства и недостатки полученных результатов и попытайтесь их улучшить. Эта операция помогает разработчику уточнить понимание требований и скорректировать процесс анализа. Мы не выступаем здесь сторонниками проектирования методом проб и ошибок. Скорее мы предлагаем проектировать, используя разумно выбранные прототипы, каждый из которых моделирует одну из частей системы, причем совокупность этих прототипов наращивает со временем свои функциональные возможности.

Конец процесса проектирования. Когда прекращать процесс проектирования? Это важный вопрос, поскольку существует опасность перегрузки системы. Архитектор редко задумывается над тем, чтобы указать на общем плане дома места прокладки трубопроводов, системы отопления или установки выключателей. Есть решения, которые нужно принимать на месте как часть процесса внедрения или, если существуют специальные требования, на нижнем уровне проектирования, выполняемом электриками или теплотехниками. Так же должно быть при проектировании систем программного обеспечения. Предлагайте только ключевые абстракции и важные механизмы, достаточные для реализации, и откладывайте на более поздний период те аспекты решения, которые оказывают незначительное влияние на видимое поведение системы.

Разбивайте большую задачу на подзадачи так, чтобы их решения были доступны специалисту в данной специфической области. В большой системе можно, например, выделить проблемы сетей баз данных или интерфейса

пользователя. Простой признак необходимости прекращения процесса проектирования состоит в том, что полученные ключевые абстракции оказываются просты и не требуют дальнейшей декомпозиции; в то же время их можно составить из существующих, повторно используемых элементов программного обеспечения.

Эволюция системы

Действия. Эволюция системы в жизненном цикле разработки объектно-ориентированного программного обеспечения совмещает традиционные этапы, включающие составление программ, их тестирование и интеграцию (комплексирование). Поэтому при объектно-ориентированном проектировании мы никогда не сталкиваемся с отдельным этапом интеграции системы. Вместо этого процесс разработки превращается в постепенное составление ряда прототипов, которые затем входят в конечную реализацию.

Пэйдж-Джонс указывает на ряд преимуществ такого метода постепенной разработки:

- * «Пользователю предоставляется обширная обратная связь тогда, когда она наиболее необходима, наиболее полезна и важна.
- * Пользователям предоставляются различные версии структур систем, позволяющие обеспечивать плавный переход от старой системы к новой.
- * Меньше возможностей отмены проекта, если он запаздывает относительно установленного срока.
- * Интерфейс главной системы тестируется в первую очередь и неоднократно.
- * Более равномерно распределены ресурсы тестирования.
- * Специалисты, занимающиеся реализацией могут видеть уже на ранних стадиях разработки результаты работы системы.
- * При нехватке времени кодирование и тестирование могут начинаться до окончания проектирования» [19].

Мы добавили бы к этому перечню следующее: составление развивающихся прототипов стимулирует разработку и оценку альтернативных решений, что позволяет получить разумный компромисс. Такая практика является обычной в других инженерных дисциплинах; к сожалению она редко встречается в программировании. При разработке системы всегда необходимо найти решение, удовлетворяющее ряду ограничений, которые соответствуют, в частности, функциональным требованиям, требованиям на сроки разработки и занимаемый объем. Определяющим является самое жесткое ограничение. Например, если критическим фактором является вес ЭВМ (при создании системы для космического корабля), необходимо учесть вес отдельных схем памяти. Разрешенный по условиям веса объем памяти ограничивает размер загружаемой программы. Если некоторое ограничение ослабляется, становятся возможными некоторые альтернативные решения; наоборот, если ограничение ужесточается, некоторые альтернативы становятся бесполезными. В процессе постепенной эволюции системы программного обеспечения мы можем определить, какое ограничение является действительно важным, а какое — нет. Оправданным подходом является проектирование, направленное в первую очередь на выполнение заданных функций и во вторую очередь на обеспечение характеристик, поскольку на ранних стадиях проектирования из-за недостатка сведений трудно понять, какая из характеристик будет «узким местом» системы. Анализируя поведение прототипов с помощью гистограммы или другим способом, разработчик может лучше понять, как доводить систему в будущем.

Изменения в процессе эволюции системы. Практика показывает, что в процессе эволюции системы в ней могут произойти следующие изменения:

- * Добавление нового класса.
- * Изменение реализации класса.
- * Изменение представления класса.
- * Реорганизация структуры класса.
- * Изменение интерфейса класса.

Каждый вид изменений вызывается различными причинами и имеет различную стоимость.

Разработчик добавляет в систему новые классы, когда вводятся новые ключевые абстракции и предлагаются новые механизмы. Цена этих изменений, связанных с перераспределением вычислительных ресурсов и расходами на управление, обычно незначительна.

Также не требует больших затрат изменение реализации класса. При объектно-ориентированном проектировании мы обычно сначала создаем интерфейс класса, а затем его реализацию. Когда интерфейс в значительной мере стабилизирован, мы можем выбрать представление данного класса и завершить составление методов. Реализация частного метода может быть снова изменена, обычно для того чтобы устранить ошибку или улучшить характеристики. Мы можем также изменить реализацию метода, для того чтобы воспользоваться преимуществами новых методов, определенных в существующих или вновь добавленных суперклассах. В любом случае изменение реализации метода не требует больших затрат, в особенности если заранее ограничен доступ к реализации класса.

Аналогичным образом можно изменить представление класса. Обычно это делается для повышения эффективности объектов класса или создания более эффективных методов. Если доступ к представлению класса ограничен, что возможно при использовании языков Smalltalk, C++, CLOS и Ada, то изменение представления не мстит логики взаимодействия объектов-пользователей с объектами данного класса (конечно, если новое представление отражает ожидаемое поведение класса). С другой стороны, если доступ к представлению класса не ограничен, что также возможно в Object Pascal, C++ и Ada, то изменение представления значительно более опасно, поскольку объекты-пользователи могут быть записаны так, что они взаимодействуют только с определенной реализацией класса. Это в особенности относится к подклассам: изменение представления суперкласса влияет на представление всех подклассов. В любом случае изменение представления класса требует затрат: нужно переработать его интерфейсы, его реализации, все объекты-пользователи (а именно, их подклассы и объекты), все объекты-пользователи его объектов-пользователей и т.д. Реорганизация структуры классов системы является обычной операцией, хотя и не такой частой, как другие изменения, о которых мы говорили выше. Как заметили Стефик и Бобров: «Программисты часто создают новые классы и реорганизуют существующие, когда замечают новые возможности для их программ» [20]. Реорганизация структуры классов обычно принимает форму изменения наследственных связей, добавление новых абстрактных классов и сдвига реализаций обычных методов в сторону высших классов в структуре. На практике реорганизация структуры классов системы особенно часто производится в начальный период, а затем, когда разработчики лучше начинают понимать взаимодействие ключевых абстракций, структура стабилизируется. Реорганизацию структуры классов следует поощрять на ранних этапах проектирования, поскольку она

может привести к значительной экономии выразительных средств, при этом нам придется создавать и поддерживать меньшее число реализаций и классов. Однако реорганизация структуры классов не проходит без затрат. Как правило, перемещение класса вверх по иерархической структуре делает ненужными все более низкие классы и требует их перетрансляции (и следовательно, перетрансляции всех классов, которые зависят от них и т.д.).

Таким же важным видом изменений, которые встречаются при эволюции системы, является изменение интерфейса класса. Разработчик обычно изменяет интерфейс класса для того, чтобы добавить ему некоторое новое свойство или операцию, которая и раньше была частью абстракции, но вначале не была учтена, а затем потребовалась объекту-пользователю. На практике, применяя эвристические методы для создания классов качества, которые рассматривались в гл. 3 (в частности, концепции создания простых, достаточных и полных интерфейсов), можно уменьшить вероятность таких изменений. Однако наш опыт показывает, что такие изменения неизбежны. Нам никогда не удавалось написать нетривиальный класс, который с первого раза имел правильный интерфейс.

Мы редко исключаем уже существующий метод; обычно это делается для того, чтобы ограничить доступ к абстракции. Чаще мы добавляем метод или вводим новый заменяющий метод, определенный в некотором суперклассе. Во всех трех случаях изменение требует затрат, поскольку оно логически воздействует на все объекты-пользователи, требуя их перетрансляции. К счастью, два последних вида изменений — добавление и замена существующего метода — являются совместимыми снизу вверх. Действительно, практика показывает, что более трех четвертей всех изменений интерфейсов, производимых при эволюции системы, являются совместимыми снизу вверх. Это позволяет применять для снижения влияния этих изменений сложные методы трансляции, такие, как пошаговая трансляция. Пошаговая трансляция позволяет перетранслировать отдельные описания и операторы вместо изменения всего модуля.

Почему так важны затраты на перетрансляцию? Они невелики для малых систем, где повторная трансляция всей программы может продолжаться всего несколько минут. По-другому обстоит дело в больших системах. Ретрансляция программы объемом в миллион строк может потребовать работы ЭВМ в течение целого дня. Можете ли вы представить ситуацию, когда необходимо изменить программное обеспечение судовой ЭВМ, и вы заявляете капитану, что он не может выйти в море из-за того, что вы производите перетрансляцию? В принципе затраты на повторную трансляцию могут быть настолько высокими, что разработчику будет запрещено вносить изменения, несмотря на то, что они дают существенные улучшения. Ретрансляция имеет большое значение при использовании объектно-ориентированных языков программирования из-за вносимых исследованием связей [21]. При использовании типовых объектно-ориентированных языков программирования затраты на перетрансляцию могут быть очень высокими; в этом случае надо искать компромисс между временем трансляции и надежностью.

Чтобы избежать ошибок при эволюции разрабатываемой системы, нужно соблюдать два правила: следить за созданием устойчивых интерфейсов и ограничивать доступ к проектным решениям, которые могут изменяться.

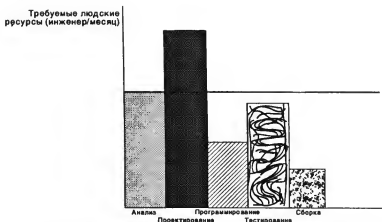


Рис. 7-3. Распределение временных ресурсов при использовании объектно-ориентированного подхода.

Модификации

Леман и Беладни сделали ряд замечаний, касающихся совершенствования программного обеспечения:

- «Программа, которая реально используется, обязательно должна изменяться, иначе она будет все менее и менее пригодной для использования (закон непрерывного изменения).
- Когда программа изменяется, ее структура становится более сложной, если при этом не предпринимаются активные усилия с целью предотвратить усложнение (закон увеличения сложности)» [22].

Поэтому мы отличаем сохранение программного обеспечения от его сопровождения. При сопровождении программного обеспечения от разработчика могут потребовать добавить новые функциональные возможности или модифицировать некоторые имеющиеся свойства. Сопровождением часто занимается другая группа специалистов, а не разработчики первоначального программного обеспечения.

Опыт показывает, что при модификации выполняются действия, несколько отличающиеся от действий, проводимых при эволюции системы. Если объектно-ориентированное проектирование с самого начала проводилось достаточно тщательно, то добавление новых функциональных возможностей или модификация некоторого существующего свойства является естественной операцией. Это подтверждается практикой даже для самых больших систем.

7.2. УПРАВЛЕНИЕ ПРОЕКТОМ

Подбор кадров

Распределение ресурсов. Мы пришли к выводу, что применение объектно-ориентированного проектирования позволяет сократить график разработки и обеспечить более высокое качество программного продукта, что помогает удовлетворить наиболее часто встречающимся требованиям к процессу

разработки программных систем. Одной из самых неожиданных сторон таких проектов оказывается возможность уменьшения общего количества требуемых ресурсов и изменение временных соотношений между разными этапами процесса. Рис. 7-3 иллюстрирует эти соотношения. По вертикальной оси отложено количество затраченных человеческих ресурсов в месяцах на одного инженера. Жирная горизонтальная линия соответствует уровню ресурсов, обычно требуемому при традиционном лавинообразном цикле разработки. При использовании объектно-ориентированного подхода ресурсы должны быть распределены иначе. На анализ они тратятся примерно в том же количестве, но на проектирование их тратятся больше, так как на этом этапе завершается большой объем работ, чем при обычном подходе. Такое увеличение не обязательно должно приводить к большему числу занятых людей; чаще оно подразумевает достаточно продолжительную работу небольшого числа хороших специалистов. Для кодирования же объектно-ориентированной работы в большей степени носит ограниченный характер. Тестирование также занимает меньше ресурсов в основном из-за того, что придание классу новых функциональных возможностей достигается главным образом за счет модификации существующего класса, поведение которого уже достаточно отработано. И наконец, интеграция системы в одно целое занимает гораздо меньше времени по сравнению с традиционными подходами, так как фактически процесс интеграции происходит на протяжении всего цикла разработки. Общая сумма человеческих ресурсов, требуемых при объектно-ориентированном подходе, обычно примерно равна или даже меньше требуемых при традиционном проектировании, и конечный продукт имеет тенденцию к улучшению.

Распределение ролей в команде разработчиков. Мы уже видели, как распределяются роли среди членов группы разработчиков программного обеспечения. При объектно-ориентированном подходе существует необходимость в четырех типах специалистов:

- * Архитекторы систем.
- * Проектировщики классов.
- * Специалисты, реализующие внутреннее строение классов.
- * Программисты-прикладники.

Архитекторы систем — это проводцы. Они относятся к числу наиболее уважаемых разработчиков и лучше других подготовлены для принятия стратегических проектных решений. Следующим наиболее уважаемым специалистам доверяется роль проектировщиков классов, ответственных за создание системных структур классов и разработку механизмов. Каждый проектировщик классов несет ответственность главным образом за создание и поддержку интерфейса какой-либо важной категории классов или подсистемы. Более молодые специалисты выступают в роли разработчиков внутреннего строения классов. Они не имеют столь широких знаний в области проектирования архитектуры классов, но знают, как правильно представить класс и реализовать это представление. Разработчики внутреннего строения классов обычно работают под непосредственным руководством проектировщика классов. Программисты-прикладники — наиболее молодые специалисты, являющиеся тем не менее экспертами в предметной области. Они берут результаты работы специалистов в области внутреннего строения классов и собирают их, используя механизмы, предложенные проектировщиками классов, для придания системе требуемых свойств.

Такое распределение труда формирует задачу подбора кадров под определенную работу; с подобными задачами приходится сталкиваться организациям, занимающимся разработкой программного обеспечения, в которых работает небольшое число специалистов высокого уровня и значительное число менее опытных программистов. К преимуществам изложенного подхода можно отнести обеспечение возможности профессионального роста для молодых специалистов, которые постоянно работают в тесном контакте со своими старшими коллегами. В процессе того, как они набираются опыта, пользуясь хорошо сконструированными классами, растет вероятность того, что со временем они смогут качественно проводить аналогичную работу.

Объектно-ориентированное проектирование делает возможным использование меньших по составу групп. Нет ничего необычного в том, что группа примерно из 30—40 человек выдает в год более миллиона строк хорошо отлаженной программной продукции. Здесь мы согласны с Бозом, который считает, что «лучшие результаты получаются в результате работы меньшего числа лучших людей» [23]. К сожалению, попытки набрать под проект меньшее количество народа, чем рекомендовано традиционным подходом, могут вызвать сопротивление. Такие попытки перечеркивают старания некоторых управленцев создать себе империю. Подобные управленцы очень любят прятаться за спину большого количества сотрудников, ведь такое подкрепление придает дополнительную уверенность в своих силах. И кроме того, в случае неудачи всегда будет на кого списать грехи.

Однако самый передовой метод проектирования — еще не повод для менеджера освобождать себя от ответственности, когда речь идет о наборе кадров: ведь небезразлично, кто это будут: думающие сотрудники или персонал, отдавший судьбу проекта на волю случая [24]. Управление — активный процесс, отнюдь не пассивный.

Этапы и результаты

Этапы. Наверно, основная причина того, что процесс объектно-ориентированного проектирования кажется менеджерам, впервые сталкивающимся с ним, чуждым для восприятия, заключается в непонимании, каким образом поступательный и итеративный процесс может обеспечить прогресс по сравнению с традиционными подходами. Прогресс можно измерять количеством времени, затраченным на завершение того или иного этапа. В этом случае мы не видим большой разницы по сравнению с существующими методами. Разница заключается в том, что понимать под успешным завершением этапа.

В процессе разработки очень полезно провести несколько официальных и гораздо большее количество неофициальных обсуждений промежуточных итогов работы. На ранних этапах экспертам, будущим пользователям и разработчикам обычно необходимо обсудить структуру прототипов будущих классов и их основные механизмы. По мере выделения ключевых абстракций и механизмов при обсуждениях начинают затрагиваться более детальные вопросы логического и физического проектирования. На этом, более позднем, этапе обсуждение включает демонстрации и анализ уже работающих прототипов.

Таким образом пользователи, менеджеры и разработчики системы получают представление о том, в каком состоянии на данный момент находится работа. Клиент может лучше чувствовать прогресс, видя работающие прототипы и обсуждая промежуточные результаты. Менеджеры могут иметь в

распоряжении те же материалы, но оценивать их, как мы увидим ниже, несколько по-другому.

Это, наверно, самое ужасное, когда менеджер оценивает степень прогресса количеством набранных строк. Данный показатель совершенно не коррелирует со степенью полноты и завершенности программы. Несмотря на все недостатки такого «неандертальского» подхода, им пользуются из-за простоты, которую он предоставляет при манипуляциях с цифрами, выливающимися в безумные показатели производительности для одних и служащими немым укором для других. Как, например, считать строки программы? По физическим строкам или по точкам с запятой? Как сравнивать несколько выражений на одной строке с одним выражением, имеющим очень большую длину? А как можно измерить затраченный труд? Считать весь персонал или только программистов? Измерять рабочий день восемью часами или добавлять все дополнительное время, которое тратит бедный инженер, просиживая на работе до утра? Традиционные методы оценки, больше рассчитанные на ранние поколения программных языков, почти не коррелируют со степенью завершенности и полноты программы и, таким образом, бесполезны применительно к объектно-ориентированным программным языкам.

Гораздо лучший способ оценки продуктивности дает опыт постоянной интеграции. Подобный эволюционный подход не предполагает наличия некоего «большого интеграционного скачка»; вместо этого прогресс измеряется качеством законченных и работающих классов, если это логическое проектирование, или модулей, если это физическое проектирование. Другой путь оценки уровня состояния работ — оценка устойчивости ключевых интерфейсов (т.е. как часто они модифицируются). Вначале интерфейсы всех ключевых абстракций меняются каждый день, если не каждый час. С течением времени наиболее важные интерфейсы стабилизируются в первую очередь, интерфейсы следующего уровня важности — во вторую очередь, и т.д. К концу цикла разработки изменений будут требовать лишь некоторые второстепенные интерфейсы; это произойдет, когда основное внимание будет уделено вопросу стыковки уже разработанных между собой классов и модулей. Иногда возникает необходимость внести некоторые изменения в конечный интерфейс, но подобные изменения обычно не вызывают цепной реакции и носят локальный характер. Но даже в этом случае их внесение должно сопровождаться детальной проработкой возможных последствий. Только затем их можно постепенно ввести в продукт как часть изменений обычного цикла модернизации, о котором мы скоро поговорим.

Часто приходится увязывать последовательный и итеративный процесс объектно-ориентированного проектирования с жесткими требованиями, налагаемыми внешними действующими лицами. Особенно часто это приходится делать, когда проект финансируется из государственных источников. Подобные трудности могут быть устранены, если заказчик с сочувствием относится к реалиям процесса разработки программного обеспечения и может позволить определенную вольность интерпретации документов, определяющих технические требования к системе. Вместо того чтобы, например, документировать ввод, обработку и вывод, мы описываем интерфейсы и поведение объектов.

Результаты. Разработка программной системы представляет собой нечто большее, чем простое написание исходных модулей. После завершения проектирования остаются еще и другие продукты, позволяющие менеджеру и пользователю полнее уяснить себе суть проекта и, что не менее важно,

иметь тем, кто будет в конечном итоге заниматься поддержкой системы, готовый банк проектных решений. Продукты объектно-ориентированного проектирования — это, как отмечено в гл. 5, диаграммы классов, модулей, процессов и объектные диаграммы. Все диаграммы основываются в конечном итоге на требованиях к программной системе. Диаграммы процессов определяют программы, которые в свою очередь являются корисными модулями в диаграммах модулей. Каждый модуль реализует соответствующую комбинацию классов и объектов, определяемых по своим диаграммам. И наконец, по диаграммам объектов создаются механизмы, соответствующие требованиям к системе, а диаграммы классов представляют ключевые абстракции, формирующие словарь предметной области.

Так как эти результаты должны рассматриваться в общем контексте, то в документации они должны быть сгруппированы определенным способом. Проект каждого значительного сегмента системы должен быть описан в отдельном документе. На практике наилучшим образом отражает логическую схему нашей системы, скорее всего, документ, имеющий следующую структуру:

- * Требуемые функции.
- * Диаграммы классов и объектов.
- * Базовые элементы классов и объектов.
- * Результаты в виде работающих прототипов.

Позднее, в процессе разработки, данный документ можно расширить, включив в него все необходимые решения, и его структура, в результате, возможно, будет выглядеть следующим образом:

- * Требуемые функции.
- * Диаграммы классов и объектов.
- * Базовые элементы классов и объектов.
- * Диаграммы модулей и процессов.
- * Базовые элементы модулей и процессов.
- * Результаты в виде работающих прототипов.

Большинство разделов данного документа могут быть написаны с помощью полуавтоматических методов.

Выпуск продукта

Дэвис и другие отмечают, что «когда процесс разработки носит последовательный характер, программный продукт сначала удовлетворяет лишь небольшому числу требований, но он создается таким образом, чтобы в дальнейшем облегчить переход к новым требованиям и, следовательно, достичь более высокого уровня адаптивности» [25]. Если встать на точку зрения конечного пользователя системы, то он хотел бы видеть ряд работающих прототипов, поступающих от организации-разработчика, каждый из которых обладал бы более высокими функциональными возможностями по сравнению с предыдущим и постепенно приближался бы к требуемой системе. С точки зрения работников этой организации, из большого числа сконструированных прототипов необходимо выбрать наиболее удачные и взять их за основу для наиболее важных интерфейсов внутри системы и, таким образом, передать заказчику наилучший вариант продукта.

При реализации масштабных проектов организация обычно выпускает новый вариант системы «для внутреннего пользования» через каждые несколько недель и затем через каждые несколько месяцев представляет за-

казкику для оценки новую систему, работающую в соответствии с требованиями проекта. В готовом состоянии такой вариант системы состоит из ряда совместимых подсистем (и их модулей), сопровождаемых также соответствующей логической и физической проектной документацией. Создание нового варианта представляется возможным, когда основные подсистемы проекта достаточно устойчивы и достаточно хорошо стыкуются между собой, чтобы можно было безболезненно обеспечить более высокий уровень функциональных возможностей.

Рассмотрим данный подход с точки зрения отдельного разработчика, ответственного за создание модулей для определенной подсистемы. Он всегда должен иметь рабочий вариант этой подсистемы. Для того чтобы успешно завершить разработку, должны иметься в наличии интерфейсы основных подсистем. Как только рабочая версия принимает устойчивые очертания, ее передают группе тестирования и интеграции, ответственной за сбор из взаимосовместимых подсистем продукта в целом. В конце концов эти собранные подсистемы должны сформировать некоторый окончательный вариант «для внутреннего пользования». Собранный вариант становится текущим прототипом системы, доступным тем разработчикам, которые хотят усовершенствовать свою часть продукта. Каждый разработчик одновременно может работать над новой версией своей подсистемы. Таким образом, работа может вестись параллельно, и устойчивость системы при этом обеспечивается работой грамотно организованных и спланированных интерфейсов между отдельными подсистемами.

Практически это означает, что в каждый момент разработки системы может существовать множество различных версий отдельных ее модулей: рабочий прототип, прототип для «внутренней» версии и, наконец, последняя рыночная версия. Это усиливает необходимость создания четкой схемы управления и контроля за состоянием различных версий. Для небольших проектов наиболее удобной единицей с точки зрения управления является программный модуль. Для более значительных проектов, которые могут состоять из десятков тысяч отдельных модулей, лучше использовать в качестве «единицы измерения» такие категории, как класс или подсистема.

Исходный текст — не единственный продукт, являющийся предметом контроля со стороны менеджера. С подобной же точки зрения могут рассматриваться и другие продукты объектно-ориентированного проектирования, например диаграммы классов, объектов, модулей и процессов.

Контроль качества

Использование объектно-ориентированного проектирования не дает права отказываться от хорошо устоявшейся практики контроля за качеством продукта. Сквозной контроль проекта все еще имеет очень важное значение для объектно-ориентированного проектирования. Но преимущество объектно-ориентированного подхода состоит в том, что изучение полученных в процессе работы диаграмм помогает уяснить структуру наиболее важных элементов проекта и сконцентрироваться на самых уязвимых местах, не увязнув при этом в разборе форматов и взаимном согласовании различных модулей. На ранних стадиях разработки контроль в основном охватывает область статических и динамических свойств ключевых абстракций и механизмов системы, отраженных в диаграмме классов и объектов. Позднее контроль охватывает главным образом физический проект системы, отраженный в диаграмме модулей и процессов.

С проблемой контроля тесно связана проблема тестирования. Следует отметить, что применение объектно-ориентированного подхода не меняет обычную практику тестирования, изменяется лишь объем тестируемых модулей. Индивидуальное тестирование классов и модулей лучше всего проводить программистам, применяющим данные классы в своих прикладных модулях. Так как большинство классов и модулей функционируют неавтономно, разработчику обычно нужно для проведения тестирования создать необходимую поддержку их интерфейса. Наверно, лучше не выбрасывать после отладки старые тесты и их результаты, так как по мере развития системы они могут пригодиться в качестве тестов, определяющих способность системы функционировать по меньшей мере так же, как раньше.

Целостное тестирование состоит из двух частей. Сначала, до окончания работы над подсистемой проектировщик классов проверяет все ее функциональные возможности. Опыт применения поддержки интерфейса для отдельных классов и модулей здесь тоже пригодится, однако на этом этапе обычно проверяется работа механизма в целом. Затем группа тестирования и интеграции, ответственная за сбор совместимых подсистем в единое целое, проводит системные тесты. Она также может использовать поддержку интерфейса для автоматической прогонки старых тестов для каждого нового варианта системы.

Итак, следует отметить, что главной задачей группы тестирования и интеграции является обеспечение политики и стандартов тестирования. Отладка модулей проводится, таким образом, программистами, применяющими эти модули и классы, тестирование подсистем выполняется проектировщиками классов, а проверка функционирования системы как целого обеспечивается группой тестирования и интеграции. Пользуясь терминами ключевых абстракций и механизмов, можно сказать, что с помощью этого подхода сначала достигается надежность прежде всего низких уровней абстракций.

Инструментальная поддержка

Работая с языками первых поколений, группе разработчиков было достаточно иметь минимальный набор инструментов: редактор, транслятор, редактор связей и загрузчик — вот часто и все, что требовалось (и скорее всего было в наличии). При определенной доле везения иногда даже можно было получить отладчик исходных текстов. Появление сложных систем полностью изменило картину: попытки создать обширную программную систему с помощью минимального набора инструментальных средств подобны попыткам построить многоэтажное здание с помощью каменных орудий.

Появление объектно-ориентированного подхода тоже в свою очередь меняет ситуацию. Традиционные средства разработки программного обеспечения могут оперировать только данными об исходных текстах, но, поскольку объектно-ориентированное проектирование выдвигает на первый план понятия ключевых абстракций и механизмов, нам необходимы инструменты, содержащие более широкий набор условных обозначений. Мы выделили по меньшей мере шесть различных типов инструментальных средств, оказывающихся полезными при объектно-ориентированном проектировании.

Типы инструментальных средств. Первым инструментом являются графические системы со встроенными элементами нотаций, представленных в гл.5. Подобный инструмент используется на самых ранних этапах процесса разработки для выработки некоторых взаимных соглашений по объектно-ориентированному анализу и проектированию, поддержке контроля за продукта-

ми разработки и координации деятельности группы. Этот инструмент также используется на протяжении всего цикла разработки, по мере того как проект доводится до вполне определенной прикладной системы. Данный инструмент также полезен для последующей поддержки системы. С его помощью, например, можно осуществлять «обратную связь» при проектировании объектно-ориентированной системы, т.е. восстанавливать по исходному тексту структуру классов и модулей проекта. Это довольно важная особенность: с традиционными средствами типа CASE разработчики могут создавать прекрасные схемы и обнаруживать, что эти схемы устаревают, как только дело доходит до применения созданных ими классов, потому что программисты начинают подгонять под себя их структуру, забывая изменять соответствующим образом проект. «Обратная связь» уменьшает вероятность того, что документация к проекту будет отличаться от его фактической реализации.

Вторым инструментом, важным с нашей точки зрения, при объектно-ориентированном проектировании, является инструмент просмотра структуры классов и модульной архитектуры системы. Иерархии классов могут оказаться настолько сложными и запутанными, что будет трудно определить абстракции, являющиеся частью проекта или кандидатами на повторное использование [26]. При просмотре фрагмента программы разработчику может понадобиться найти определение класса, которому принадлежит тот или иной объект. Найдя этот класс, он возможно захочет совершить экскурсию туда, где определен тот или иной суперкласс. Просматривая суперкласс, разработчик может выразить желание увидеть все примеры его использования, прежде чем приступить к изменению его интерфейса. Подобный просмотр может оказаться весьма сложным занятием, если при этом приходится заботиться о файлах, которые относятся к физической структуре системы, а не к логической. Поэтому инструмент просмотра является таким важным при объектно-ориентированном проектировании. Smalltalk, например, позволяет просматривать классы системы именно описанным нами способом. Подобные же возможности существуют, правда в различной степени, во всех программных языках, которые мы используем в этой книге.

Третий с нашей точки зрения важный, если не определяющий, инструмент — это пошаговый транслятор. Как мы выше отметили, тот эволюционный тип разработки, который определяется объектно-ориентированным подходом, с необходимостью влечет за собой появление пошагового транслятора, обеспечивающего трансляцию отдельных операторов и функций. Мейрович считает, что «ОС Юникс в том виде, в каком она сейчас находится (с ориентацией на пакетный режим трансляции больших программных файлов и переводом их в библиотечные модули, которые затем объединяются с другими модулями), не обеспечивает необходимую поддержку для объектно-ориентированного программирования. Совершенно неприемлемо тратить 10 мин. на трансляцию и редактирование только для того, чтобы изменить один из методов, и тратить целый час на трансляцию и редактирование, чтобы добавить одну переменную в структуру суперкласса высокого уровня! Пошагово-транслируемые методы и пошагово-транслируемые описатели необходимы для быстрой отладки» [27]. Пошаговые трансляторы существуют для большинства языков, описанных в этой книге, однако большинство их реализаций содержит традиционные трансляторы, ориентированные на пакетный режим.

Кроме того, почти для всех истривнальных проектов необходимо наличие отладчиков, понимающих семантику классов и объектов. При отладке часто требуется проверить значение переменной объекта и переменной клас-

са, связанной с объектом. Традиционные отладчики для обычных программных языков не содержат знания о классах и объектах. Пытаясь, таким образом, использовать стандартный Си-отладчик для программ, написанных на C++, мы, даже если это нам удастся, не можем получить по-настоящему важную информацию для отладки объектно-ориентированной программы. Наиболее тяжелая ситуация складывается с объектно-ориентированными языками, поддерживающими мультипрограммный режим работы компьютера. В каждый данный момент работы такой программы может существовать несколько активных процессов. В подобных случаях требуется отладчик, позволяющий разработчику осуществлять контроль за каждым из процессов, обычно по схеме «объект-за-объектом».

При реализации больших проектов необходимо также следить за их конфигурацией и иметь инструмент контроля за версиями. Как было выше замечено, для небольших проектов наилучшей единицей контроля может служить отдельный модуль; для более сложных проектов наиболее приемлемой единицей является подсистема.

Последний инструмент, на котором мы остановим внимание при рассмотрении объектно-ориентированного проектирования, — это библиотекарь классов. Большинство языков, которыми мы пользуемся в этой книге, имеют заранее определенные библиотеки классов. По мере доводки проекта эти библиотеки растут, так как к ним с течением времени добавляются модули, созданные разработчиком. Через довольно короткое время такая библиотека может вырасти до огромных размеров, и в результате разработчик скорее всего встретится со значительными трудностями при попытке найти тот или иной нужный ему модуль. Одной из причин большого размера библиотеки может стать многократная реализация того или иного класса, имеющая каждая свою семантику. Если примерная цена (обычно завышенная) нахождения определенной компоненты выше, чем примерная цена (обычно заниженная) создания заново такой компоненты, тогда любую надежду на ее повторное использование можно считать потерянной. Поэтому хорошо иметь хотя бы самый примитивный библиотекарь, позволяющий разработчику сортировать классы и модули по определенному принципу и добавлять в библиотеку полезные классы и модули по мере их создания.

Организационные факторы. Чтобы создать мощный инструмент разработки программного обеспечения, в его структуру должны быть включены библиотекарь и инструментальный редактор. Задачей библиотеки является организация библиотеки классов проекта. Если не прилагать активных усилий, такая библиотека может стать громадной свалкой всякого барахла — ненужных классов, в которых неохота копаться ни одному из разработчиков. Часто также необходимо подтолкнуть людей к более активному использованию уже существующих наработок, а библиотекарь может облегчить процесс повторного использования, рассортировав в удобном порядке результаты текущего проекта. Задача инструментального редактора — создание проблемно-ориентированных и модернизация существующих инструментов под определенный проект. В процессе работы, например, могут потребоваться инструменты типа замесителя ввода-вывода для тестирования пользовательского интерфейса или своего собственного словаря классов. Инструментальный редактор наилучшим образом сможет помочь при создании подобных вещей, составляя их, как правило, из существующих модулей, принадлежащих библиотеке классов. Затем эти инструменты можно также использовать и в других проектах.

Менеджер, который уже сталкивался с проблемой ограниченности людских ресурсов, может посоветовать на то, что подобные мощные инструменты, а также библиотекарь и инструментальный редактор — непозволительная роскошь для них. Мы не отрицаем, такова реальность, когда дело касается некоторых проектов с ограниченными ресурсами. Однако оказывается, что во многих проектах работы в этом направлении ведутся, правда, обычно они рассматриваются как некий «довесок» к основной работе. Мы придерживаемся той точки зрения, что те прямые финансовые вложения в людей и в соответствующий инструментарий, которые придают подобной дополнительной деятельности более целенаправленный характер и повышают ее эффективность, в конце концов окупаются.

7.3. ДОСТОИНСТВА И НЕДОСТАТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Достоинства объектно-ориентированного проектирования

Новые технологии, и в частности объектно-ориентированную технологию, обычно выбирают по одной или по двум причинам. Во-первых, из-за желания вырваться вперед на рынке (за счет меньшего времени разработки, большей гибкости продукта, или лучшей предсказуемости хода работ) с помощью новых технических средств, обеспечивающих громадный финансовый эффект. Во-вторых, столкнувшись со сложными проблемами, решить которые можно, по-видимому, лишь обратившись к новым технологиям. И если первые пользователи новых технологий произнесут затем достаточное количество хвалебных слов в их адрес, то и другие, менее склонные к риску организации последуют примеру первооткрывателей.

Объектно-ориентированное проектирование не является новой технологией, но в то же время она не является и достаточно привычной в основном из-за того, что гораздо меньшее количество людей имеют опыт работы с ней, чем, например, со структурным проектированием. Мы, естественно, ожидаем, что ситуация постепенно изменится. Тем временем все большее количество людей уже слышало об успехах первых разработчиков, взявших на вооружение этот метод. Но, честно говоря, мы понимаем, что люди должны «на своей шкуре» убедиться в способности метода решить задачу разработки сложной программной системы. А пока вам придется принять на веру то, что при использовании объектно-ориентированного подхода можно добиться значительных результатов.

В гл. 2 мы показали, что использование объектной модели ведет к созданию комплексов, имеющих все свойства хорошо структурированных сложных систем. Объектная модель создает умозрительную схему объектно-ориентированного проекта и, таким образом, все ее преимущества связаны с объектно-ориентированным методом. Мы также назвали следующие преимущества применения объектной модели (и, таким образом, объектно-ориентированного проектирования):

- * Использование выразительных средств объектных и объектно-ориентированных языков.
- * Поддержка повторного использования отдельных составляющих программного обеспечения.
- * Создание более открытых систем.
- * Снижение риска при разработке.
- * Активизация познавательных способностей человека.

Дополнительное осмысление этих преимуществ приводит к новым выводам; в частности, они указывают на то, что объектно-ориентированный подход может уменьшить время разработки и конечный размер исходных текстов.

Пока еще нет достаточного количества примеров для оценки эффективности объектно-ориентированного программирования при разработке формальных экономических моделей типа CoCoMo или Price-S.

Недостатки объектно-ориентированного проектирования

Недостатки объектно-ориентированного проектирования могут повлиять на:

- Характеристики системы.
- Начальные затраты.

Ухудшение характеристик. Существует определенная плата за посылку сообщения от одного объекта к другому, выражающаяся в некотором ухудшении быстродействия. Как мы говорили в гл. 3, для тех обращений к методам, которые не могут быть реализованы статически, необходимо провести динамический поиск, чтобы найти соответствующий метод, определенный в классе, которому принадлежит объект, получающий сообщение. Опыт показывает, что обращение к методу может занимать в 1,75-2,5 раза больше времени, чем обращение к обычной подпрограмме [29, 30]. Но что значит «не могут быть реализованы статически»? Как правило, динамический поиск оказывается нужен, примерно, в 20% случаев от общего числа обращений. Поэтому транслятор в хорошо типизированном языке часто может сам определить какие вызовы могут быть статически реализованы, и написать код вызова подпрограммы, а не поиска метода.

Другой недостаток связан главным образом даже не с особенностями объектных и объектно-ориентированных языков, а со способами их применения в контексте объектно-ориентированного проектирования. Мы много раз говорили, что объектно-ориентированное проектирование приводит к созданию систем, составленных из абстракций разных уровней. Особенностью такой структуры является сравнительно небольшой размер отдельных методов, так как они в свою очередь составлены из методов, соответствующих более низким уровням абстракций. Другая особенность такого разбиения на уровни — необходимость создания отдельных методов для достижения доступа к защищенным переменным объекта. Многочисленность методов ведет к излишнему количеству вызовов. Вызов метода высокого уровня абстракции обычно приводит к тому, что по системе проходит каскад вызовов; методы высоких уровней вызывают методы более низких и т.д. Для программ, в которых время является ограничивающим фактором, большое число вызовов может оказаться неприемлемым. Но, с другой стороны, такое разбиение на уровни необходимо для понимания системы; во многих случаях просто невозможно создать сложную систему, не прибегнув к разбиению на уровни. Мы рекомендуем сначала получить нормальное функционирование, а затем определить уже в работающей системе те места, в которых существует опасность временного запаздывания. Впоследствии в систему можно внести соответствующие исправления, переопределив некоторые методы как встроенные (увеличив, таким образом, быстродействие за счет увеличения объема программы), сняв защиту с некоторых переменных или в крайнем случае, переделав объект в обычную запись вместо реализации класса.

Подобное ухудшение характеристик связано с вложенностью классов: класс, находящийся на самом конце линии наследования может иметь множество суперклассов, коды которых должны быть присоединены при редактировании этого отдельного класса. Для небольших проектов можно посоветовать в данном случае не использовать глубокие иерархии классов, так как для этого потребуются присоединять слишком большое число объектных кодов. Проблему можно частично решить, используя такие транслятор и редактор связей, которые удаляют ненужные коды.

Следующий недостаток, связанный с применением объектных и объектно-ориентированных программных языков, вытекает из особенностей структуры выполняемых прикладных программ. Большинство трансляторов размещают объектные коды по сегментам, причем коды каждой программной единицы (обычно это файл) размещаются в одном или в нескольких сегментах. Такая модель предполагает высокую степень локальности вызовов: программы внутри одного сегмента вызывают подпрограммы из того же сегмента. Однако в объектно-ориентированных системах редко можно добиться подобной локальности ссылок. В больших системах разные классы, как правило, бывают объявлены в разных файлах и, так как методы каждого класса обычно строятся с помощью методов, принадлежащих другим классам, вызов одного метода обычно требует привлечения содержания многих сегментов. Это нарушает предположения, заложенные в компьютеры и касающиеся синхронизации работы программ (особенно для систем с виртуальной памятью). Но, с другой стороны, мы ведь специально решили разделить проектные решения на локальные и физические. Если система выходит из строя в процессе работы из-за слишком интенсивного обмена сегментами, то решение проблемы можно найти, изменив физическое расположение классов и разместив их по другим модулям. При этом мы изменим физическую модель системы, что не окажет влияния на ее логическую модель.

Наконец, последний возможный недостаток объектно-ориентированных систем связан с динамическим размещением и уничтожением объектов. Подобное размещение отличается от статического, осуществляемого либо глобально, либо внутри стека. Для многих типов систем динамическое размещение не вызывает трудностей, но для задач с ограниченным ресурсом времени мы часто не можем выделить достаточного времени для завершения всех циклов, необходимых для динамического размещения. Существует простое решение этой проблемы: мы рекомендуем завершать динамическое создание таких объектов как часть создания программы, а не во время выполнения алгоритмов с ограниченным ресурсом времени.

Тем не менее достоинства объектно-ориентированных систем, как правило, перевешивают все перечисленные недостатки. Руссо и Каплан, например, сообщают, что быстрдействие программы, написанной на C++, часто выше, чем у аналогичной программы с теми же функциональными возможностями, написанной на языке Си [31]. Они полагают, что это связано с использованием виртуальных функций, которые в некоторых случаях устраняют необходимость прямой проверки типов данных и структур. Опыт также показывает, что размер исполняющих модулей объектно-ориентированных систем обычно меньше, чем у функционально эквивалентных им систем, созданных с помощью традиционных методов.

Начальные затраты. Для некоторых проектов начальные затраты, связанные с внедрением объектно-ориентированного подхода, могут оказаться достаточно высокими. Использование любой подобной технологии требует

вложения капиталов в инструменты для разработки программного обеспечения. Кроме того, если организация-разработчик впервые использует тот или иной объектный или объектно-ориентированный программный язык, обычно отсутствуют необходимые проблемно-ориентированные библиотеки. Им приходится либо начинать все с нуля, либо как-то стараться соединить имеющиеся неориентированные модули с объектно-ориентированными. Наконец, первый опыт использования объектно-ориентированного подхода обречен на неудачу без соответствующего предварительного обучения. Объектный, объектно-ориентированный язык программирования — это не просто «еще один язык программирования», который можно изучить на трехдневных курсах или прочитав одну книжку. Требуется время для выработки правильного понимания предмета объектно-ориентированного проектирования, причем соответствующий тип мышления должен быть принят и разработчиками, и менеджерами.

Переход к объектно-ориентированному проектированию

Изменение способа мышления является важным моментом. Кемпф, например, считает, что «изучение объектно-ориентированного программирования может оказаться гораздо более трудной задачей, чем изучение «просто» обычного языка программирования. Причиной тому является скорее различие в стилях программирования, чем отличие синтаксисов при сохранении самой структуры языка. В данном случае приходится иметь дело не с новым языком, а с новым типом мышления» [32].

Так как же перейти к объектно-ориентированному типу мышления? Мы рекомендуем следующее:

- * Обеспечить соответствующее обучение разработчиков и менеджеров.
- * Сначала использовать объектно-ориентированный подход при разработке наименее рискованных проектов и заставить разработчиков не бояться совершать при этом ошибки; использовать затем тех, кто работал в этой группе в качестве инициаторов новых проектов и экспертов в области объектно-ориентированного подхода.
- * Знакомить разработчиков и менеджеров с примерами хорошо структурированных объектно-ориентированных систем.

Хорошим новым проектом может стать создание какого-либо инструмента разработки программного обеспечения или проблемно-ориентированных библиотек классов, которые затем могут быть использованы в других проектах. Наш опыт показывает, что разработчику-профессионалу требуется всего несколько недель чтобы овладеть синтаксисом и семантикой нового языка. Чтобы начать понимать важность и полезность понятий класса и объекта, тому же разработчику понадобится дополнительно еще несколько недель. Но чтобы стать квалифицированным проектировщиком классов, разработчику, возможно, придется потратить целых шесть месяцев. И это не всегда плохо, ведь в любой дисциплине требуется время, чтобы овладеть всем ее искусством.

Мы считаем, что обучение на примерах часто является наиболее эффективным из всех подходов. Если организация накопила некоторую критическую массу различных примеров применения объектно-ориентированного подхода, привлечение новых разработчиков и менеджеров и обучение их объектно-ориентированному проектированию становится гораздо более простым делом. Разработчики начинают с создания программ, в которых они

используют уже готовые, хорошо организованные абстракции. С течением времени те разработчики, которые хорошо разобрались и активно использовали эти компоненты программного обеспечения под наблюдением более опытных специалистов, набираются достаточного опыта, чтобы самостоятельно начать разработку сложных концептуальных структур для объектных модулей и стать, таким образом, проектировщиками классов. Рассмотрим теперь некоторые примеры в гл. 8-12, иллюстрирующие практическое применение объектно-ориентированного проектирования, реализованные с использованием различных языков и затрагивающие независимые друг от друга предметные области.

Заключение

- * Цикл разработки программного обеспечения с использованием объектно-ориентированного проектирования представляет собой последовательный итеративный процесс создания системы.
- * Структурный анализ является привлекательным инструментом создания исходного продукта для объектно-ориентированного проектирования, однако объектно-ориентированный анализ оказывается еще более эффективным инструментом.
- * Проектирование можно начинать, если существует (возможно исполная) формальная или неформальная модель решения задачи; оно заканчивается там, где на первый план начинают выходить вопросы компоновки системы, а не разложения ее на составляющие.
- * В объектно-ориентированном проектировании невозможен «большой интеграционный скачок» (сборка системы за короткое время из разрозненных элементов).¹⁾
- * В процессе эволюции системы предусмотрено несколько переходов ее в новое качество; каждое такое изменение требует различных затрат.
- * Использование объектно-ориентированного проектирования меняет практику управления и, в частности, затрагивает вопросы подбора кадров, этапности, выпуска продукта и его дальнейших версий, контроля качества и инструментальной поддержки.
- * Объектно-ориентированное проектирование имеет много достоинств и некоторые недостатки; опыт показывает, что достоинства метода намного превосходят его недостатки.
- * Организационный переход к использованию объектно-ориентированного проектирования требует изменения самого типа мышления; изучение объектного или объектно-ориентированного языка представляет собой нечто большее, чем изучение «еще одного языка программирования».

Дополнительная литература

Использование методов структурного анализа в предварительной работе перед использованием объектно-ориентированного проектирования описано в работах Alabios [F, 1988] и Stark [F, 1986]). Использование с той же целью методов объектно-ориентированного анализа описано в работах Baillin и Moore [B, 1987], Gernosek, Monterio и Pribyl [B, 1987], Dahl [B, 1987], Mellor, Hecht, Tryon и Smith [B, 1987]. Работа Aron [H, 1974]) представляет собой обстоятельный обзор проблем управления как программистом, так и командой программистов. Для настоящего понимания того, что происходит при разработке сложных систем, когда прагматические соображения заставляют забыть о теории, ознакомьтесь с работами Glass [G, 1982], Lammers [H, 1986] и Humphrey [H, 1989]. Кроме того, DeMarco и Lister [H, 1987] и Yourdon [H, 1989] предлагают ряд очень полезных рекомендаций менеджеру, под руководством которого ведется разработка сложных программных систем. Предложения по тому, как перевести отдельных специалистов или целые организации на использование объектных моделей, можно найти в книгах: Goldberg [C, 1987], Goldberg и Kay [G, 1977] и Kempf [G, 1987]. Работа Vonk [H, 1990] представляет собой обстоятельный учебник по прототипированию.

¹⁾ Сборка системы за короткое время из разрозненных элементов. — Прим. перев.

Часть III

ПРИМЕНЕНИЯ

Создание какой-либо теории подразумевает наличие достаточных знаний об основных сторонах изучаемого предмета. Что касается проблемы компьютеризации, то таких знаний у нас недостаточно, и мы не имеем возможности подойти к предмету абстрактно.

В этой ситуации нам не остается ничего другого, как внимательно изучать отдельные, хорошо понимаемые практические примеры, чтобы на основе результатов перейти к рассмотрению более общих принципов.

Марвин Мински (Marvin Minsky)

«Форма и содержание компьютерной науки»

Глава 8

Smalltalk. Система домашнего отопления

С точки зрения инженера, даже самая стройная теория является бесполезной, если она не отвечает на вопрос о том, как практически создавать системы для решения реальных задач. В данной части книги рассматриваются практические приемы создания программных систем на основе методологии OOD, приведены примеры того, как от исходных требований к той или другой системе на основе использования терминологии, обозначений и процедуры OOD осуществляется переход к реализации данной системы. В качестве примеров выбраны задачи из различных предметных областей. Каждая задача имеет собственную специфику и присущие только ей проблемы. В каждом примере используется свой язык программирования, чтобы проиллюстрировать применимость методологии OOD для всего спектра объектных и объектно-ориентированных языков. В данной главе использован язык Smalltalk. Поскольку главным предметом нашего рассмотрения является процесс проектирования, а не программирования, тексты программ несколько сокращены. Однако основные детали перехода от проекта к его реализации и особенности использования конкретного языка программирования приведены достаточно подробно.

8.1. АНАЛИЗ

Определение границ предметной области

Приведенный выше текст содержит основные требования к системе отопления. При чтении этих требований можно задать следующие вопросы: Нужно ли моделировать процесс передачи тепла через стены дома? Есть ли

Требования к системе домашнего отопления

Изложенные ниже требования к системе были первоначально сформулированы Уайтом [1], а затем уточнены Кертон [2]. Блок-схема системы отопления показана на рис. 8-1. Основная функция рассматриваемой системы состоит в регулировании теплового потока в каждой из комнат дома для поддержания требуемой рабочей температуры t_w . Рабочая температура для каждой комнаты вычисляется исходя из заданной желаемой температуры t_d (которую пользователь задает через систему ввода) с учетом наличия в той или иной комнате людей. Если в какой-либо комнате люди отсутствуют, температура в ней поддерживается на 5° (по Фаренгейту) ниже желаемой. Кроме того, система хранит информацию о недельном цикле использования комнат и за 30 мин до ожидаемого появления в конкретной комнате людей начинает повышать в ней температуру до нужного уровня. Если в течение двух недель подряд установленный ранее цикл работы нарушается, система корректирует свой график. В каждой комнате дома имеются датчик для измерения температуры и инфракрасный датчик для непрерывного определения присутствия в комнате людей. Для управления системой и контроля за ее работой используется специальный интерфейс пользователя, который позволяет вводить следующие команды:

- * Включение обогревателя Переключатель, позволяющий включать и отключать обогреватель системы
- * Установка желаемой температуры Устройство для установки желаемой температуры (в каждой комнате)

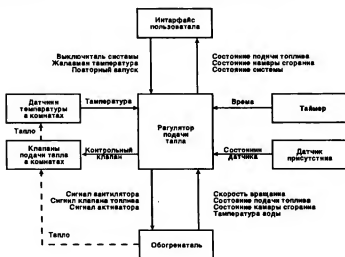


Рис. 8-1. Блок-схема системы отопления.

* Переключатель-индикатор перезапуска при сбое

В системе имеются также средства индикации:

* Индикатор состояния

* Переключатель-индикатор перезапуска при сбое

Комбинированный переключатель-индикатор, позволяющий пользователю перезапустить систему после сбоя

Служит для индикации состояния системы обогрева (функционирует/не функционирует)

Комбинированный переключатель-индикатор, который автоматически выключается при обнулении сбоя в подаче топлива или в работе камеры сгорания

Непрерывный отсчет текущего времени с дискретностью в 1 с обеспечивает таймер. Перенос тепла осуществляется с помощью циркуляции горячей воды, нагреваемой в обогревателе. В каждой комнате имеется специальный клапан для регулирования подачи нагретой воды, который может быть дистанционно закрыт или открыт (полностью). Обогреватель состоит из котла, клапана для подачи топлива, воспламенителя, вентилятора и датчика температуры воды. Нагреваемая в котле вода циркулирует по комнатам через управляемые клапаны. Обогреватель может находиться либо в активном, либо в пассивном состоянии (выключаться) в зависимости от потребности в тепле. Процедура включения обогревателя выполняется в следующем порядке:

- * включается двигатель вентилятора;
 - * система контролирует обороты двигателя и при достижении заданной величины (об/мин) открывается топливный клапан и зажигается топливо;
 - * при достижении заданной температуры воды в котле открываются клапаны подачи воды в комнаты;
 - * включается индикатор состояния работы обогревателя.
- Включение выполняется в следующем порядке:
- * закрывается клапан подачи топлива и через 5 с останавливается двигатель вентилятора;
 - * отключается индикатор состояния;
 - * закрываются все клапаны подачи воды в комнатах.

Работа обогревателя контролируется датчиком подачи топлива и оптическим датчиком камеры сгорания. При обнаружении каких-либо отклонений выполняется выключение обогревателя: выключается переключатель-индикатор и закрываются все клапаны подачи воды. Для определения необходимого количества тепла и управления его подачей в комнаты служит регулятор теплового потока, взаимодействующий с другими компонентами системы. Регулятор теплового потока управляет процессом поддержания рабочей температуры и графиком ее изменения независимо от работы обогревателя. Если температура в каком-либо помещении опускается на два градуса ниже требуемой или поднимается на два градуса выше требуемой, то формируется команда на подачу или прекращение подачи тепла в это помещение. При включенных переключателях (обогревателя и перезапуска) и необходимости подачи тепла в какую-либо комнату производится включение обогревателя (если она была выключена). Если потребность в подаче тепла исчезает, обогреватель выключается. То же происходит при отключении любого из двух переключателей. Повторное включение обогревателя после остановки не может быть выполнено ранее чем через 5 мин.

какие-либо требования по безопасности, связанные с наличием высоких температур? Что нужно делать в случае неисправности клапанов подачи воды или датчиков температуры? Мы стараемся сопоставить решаемую задачу с другими аналогичными задачами, для которых уже существует решение. Например, можно отметить, что система домашнего отопления напоминает систему тепличного хозяйства (гл. 5), но существенно отличается от системы регулирования (гл. 7). В результате у нас имеются все основания заключить, что система отопления относится к категории задач управления процессом, но не к категориям баз данных или научного эксперимента, хотя элементы двух последних в нашей задаче тоже присутствуют.

Уточнение требований. Внимательно прочитав требования, мы можем обнаружить, что они являются неполными, излишне подробными и внутренне противоречивы. В то же время эти требования реальны, не выдуманы. Даже если мы будем иметь исчерпывающую информацию о поставленной проблеме, мы не сможем в полной мере использовать эти знания из-за несовершенства наших инструментальных средств. Сказанное выше не является критикой каких-либо конкретных требований, это только подтверждение известного факта, основанного на практическом опыте: при выполнении проекта всегда приходится мириться с определенными неясностями.

Что касается неполноты изложенных требований к системе, то можно привести несколько конкретных примеров. В частности, не указан масштаб системы отопления: контроль температуры может вестись и в 10, и в 10 000 комнат. Является ли используемое оборудование одно- или многопроцессорным? Если процессор один, то насколько мы ограничены ресурсами памяти и производительности? Для простоты предположим, что системные проектировщики установили требование по реализации программной поддержки на основе одного процессора и его ресурсы достаточны для управления системой при заданном числе комнат. Разумеется, желательно спроектировать систему так, чтобы она легко адаптировалась к любому числу комнат при минимуме вносимых изменений.

В требованиях не говорят о том, как система должна реагировать на параллельные события, когда люди появляются одновременно в двух комнатах. Система домашнего отопления не относится к числу задач с жесткими ограничениями при работе в реальном времени, в требованиях определены критические интервалы времени в несколько секунд, но не милли- и не микросекунд. Можно предположить альтернативные подходы к проекту с различным распределением ресурсов памяти и времени. Для создания иллюзии параллелизма при наличии одного процессора необходимо обеспечить время реакции на события в пределах зрительной реакции человека. Если мы предпочитаем систему, управляемую событиями, необходимо обеспечить работу процессора в режиме прерываний по сигналам аппаратных средств. Однако на данном этапе нет срочной необходимости делать подобный выбор, его можно отложить до того момента, когда будет полная информация, чтобы взвесить каждое решение. Излишне подробно изложены требования к регулятору потоков тепла. В частности, определено, что он отслеживает недельный график использования комнат. Полезность функции отслеживания графика сомнений не вызывает, но непонятно почему эту функцию должен выполнять регулятор. Предъявляемые исходные требования не должны в общем случае затрагивать особенности проектирования, поскольку это ограничивает свободу программиста, заставляет отказываться от более оптимальных вариантов, которые могут упростить решение всей задачи. Будем рассматривать предъявленное к регулятору требование в свободной интерпре-

тации и условимся, что главное — обеспечить функцию отслеживания недельного графика. Механизм реализации этой функции может отличаться от изложенного в требованиях.

Наличие в требованиях противоречий также не вызывает особого удивления: чем сложнее система, тем вероятнее в ней столкновение различных требований. Изложенные нами требования в этом смысле не являются исключением. Проанализируем внимательно требования к переключателю-индикатору сбоев. С одной стороны, установлено, что он автоматически отключается при сбое в подаче топлива или в камере сгорания. С другой стороны, утверждается, что отдельные датчики могут передавать в систему сигналы об отклонениях, ведущих к выключению обогревателя, выключению переключателя-индикатора и закрытию клапанов. В первом случае, таким образом, говорится о непосредственном отключении переключателя-индикатора, а во втором сначала выключается обогреватель и лишь затем отключается переключатель-индикатор. Приходится выбирать, на каком варианте поведения системы остановиться. Если предположить, что предварительного выключения обогревателя не требуется, то оператор будет оповещен о неисправности сразу при ее обнаружении, а выключение обогревателя задержится во времени. В другом варианте информация о неисправности будет задержана, но обогреватель выключается сразу. Чтобы разобраться в такой ситуации, лучше всего спросить мнение пользователя или показать пользователю альтернативные прототипы и предложить сделать выбор. Поскольку важнее чаще всего безопасность системы, остановимся на варианте с немедленным выключением обогревателя.

Формирование словаря предметной области. Так как мы проектировали, ни пользователь не могут заранее иметь полное представление о всех подробностях поведения системы, следует воспользоваться методом последовательного приближения. Мы будем создавать варианты прототипов системы на языке Smalltalk и демонстрировать их работу заказчику. Когда мы достигнем согласия (возможно, изменив при этом некоторые требования), то модифицируем систему в части обеспечения интерфейса с внешним оборудованием (датчиками, клапанами), сохранив ее принципиальную основу.

Теперь полезно зафиксировать наиболее существенные факты относительно предметной области. На основе структурного анализа мы в первую очередь создадим эскиз диаграммы связей и в соответствии с ней диаграмму потоков данных самого верхнего уровня. Более подробно структурный анализ рассматривается в гл. 10, поскольку для решаемой в данной главе задачи исходные требования достаточно хорошо определены, а сама задача относительно статична. Кроме того, все мы являемся в определенной степени специалистами по данной предметной области: в наших домах и по месту работы существуют те или другие отопительные системы.

Мы назвали текущий этап анализом, но на самом деле это уже начало последовательно-возвратного проектирования. В гл. 7 было показано, что объектно-ориентированная методология имеет достаточно хорошую систему документирования знаний о ключевых абстракциях предметной области.

На рис. 8-2 показана схема объектов, отражающая верхний уровень структуры системы. На схеме отражено наличие в системе множества объектов, упомянутых выше при рассмотрении требований к системе: комнаты, регулятор, обогреватель и интерфейс пользователя.

В требованиях указан только один вид интерфейса пользователя, а в действительности существуют две категории пользователей: люди, находящиеся в комнатах, и операторы системы. По причинам, которые будут объяс-



Рис. 8-2. Схема объектов домашней отопительной системы.

нены ниже, на верхнем уровне системы мы рассматриваем только интерфейс оператора. Обратим внимание, что на схеме показана только одна внешняя связь между объектами. Эта связь отражает поток тепла от печки в комнаты для регулирования температуры. В системе существует обратная связь, находящаяся вне нашей предметной области. Мы не будем заботиться о наличии других источников обогрева или охлаждения (например, камины или открытые окна).

Другие отношения между объектами на схеме этого уровня не показаны, поскольку на текущем этапе анализа это преждевременно. Когда мы перейдем к процессу проектирования, данную схему необходимо изменить и указать большее количество подробностей, что позволит уточнить границы каждого объекта и принять оптимальные проектные решения.

Мы уже отметили выше, что функция отслеживания недельного графика обогрева является одной из основных в системе, но не обязательно должна выполняться регулятором. Зафиксируем это решение, приведя схему объектов, означающих помещение дома (рис. 8-3). В каждом помещении имеются клапан подачи воды, датчики измерения и задания температуры, датчик присутствия людей. Кроме того, для каждого помещения существует свой график посещения. Размещение данных о графике посещения в каждом объекте-помещении (а не концентрация их всех в регуляторе) объясняется тремя причинами.

Во-первых, отнесение этих данных к регулятору приведет к тому, что при увеличении числа помещений придется корректировать эти данные и, следовательно, процесс развития системы усложняется. Во-вторых, сведения о графике посещения индивидуальны для каждой из комнат и размещение их в соответствующих объектах увеличивает общность абстракций. В-третьих, принятое решение делает регулятор более автономным. Задача регулятора состоит лишь в том, чтобы управлять подачей тепла в помещение, а помещение «само» должно решить, нужно ли ему тепло в зависимости от того, находятся или ожидаются в нем люди (по собственному графику), а также с учетом разности между фактической и желаемой температурой. Так же как на предыдущей схеме, отношения между объектами на рис. 8-3 не показаны. На данном этапе эта информация опускается.



Рис. 8-3. Схема объектов-помещений.

Анализ предметной области

Теперь, имея в общих чертах модель создаваемой системы, можно перейти к анализу предметной области, изложенному в гл. 4.

Реальное состояние задачи. Исходя из требований к системе, мы выделили следующие ключевые абстракции:

Система отопления	Таймер	Дом
Помещение	График посещения	Датчик текущей температуры
Датчик присутствия людей	Клапан подачи воды	Датчик желаемой температуры
Интерфейс оператора	Выключатель отопления	Переключатель-индикатор повторного включения
Индикатор работы печки	Печка	Датчик температуры в котле
Клапан топлива	Вентилятор	Воспламенитель
Регулятор подачи тепла		

Существует также абстракция температуры, но она не представляет собой реально осязаемый предмет, а является свойством, которое можно измерить и почувствовать. Температура, несомненно, является существенным элементом словаря предметной области. Установив, что температура в системе измеряется в градусах по Фаренгейту, и вспомнив, что в системе используется вода для переноса тепла, мы сможем сделать логичные предположения относительно ожидаемого диапазона температуры. Вода не должна иметь температуру ниже точки замерзания и выше перегретого пара.

Из числа объектов, указанных в требованиях, мы оставили в стороне пользователей системы. Люди рассматриваются в данном случае вне системы как посредники, являющиеся причиной ряда событий, регистрируемых системой отопления. С точки зрения программы они никак не отражены. Однако заметим, что нас интересуют два вида пользователей: операторы системы и простые посетители помещений. Их отношение к системе принципиально различается: одни из них (посетители) управляют значением желаемой тем

пературы в комнатах, а другие (операторы) могут включать, выключать и перезапускать систему. Исходя из этого, интерфейс оператора следует отнести к абстракциям верхнего уровня, отделив его от интерфейса обычных посетителей.

События. Перечислим теперь внешние события, на которые система отопления должна реагировать:

- * Посетитель изменил значение желаемой температуры.
- * Посетитель вошел в помещение или вышел из него.
- * Изменилась фактическая температура в помещении.
- * Неисправность в подаче топлива или в камере сгорания.
- * Оператор включил или выключил отопление.
- * Система отопления включена или отключена.
- * Оператор выполнил повторный запуск системы после сбоя.

Все события асинхронны; определено только одно событие, совершаемое периодически:

- * Сигнал таймера с интервалом в 1 с.

Это событие не является внешним, но с ним связано управление вентилем и котлом.

Перечислив эти события, мы можем сопоставить их с ключевыми абстракциями, определив тем самым границы системы. Можно, например, предположить, что за временем следит таймер, а такие компоненты, как график посещения, только реагируют на сигналы таймера. Выявление внешних событий позволяет также уточнить, что система делать не может. В частности, у нас нет средств обнаружения отказов клапанов подачи воды и, следовательно, система не может реагировать на такие неисправности.

Перечень внешних событий помогает контролировать полиоту списка ключевых абстракций: должна существовать хотя бы одна абстракция, связанная с каждым событием. Например, отметим, что в исходном списке ключевых абстракций нет объектов, позволяющих обнаруживать отклонения в подаче топлива и в работе камеры сгорания. Это заставляет дополнить список еще двумя ключевыми абстракциями: датчиком неисправности подачи топлива и датчиком сбоя в камере сгорания.

Документирование ключевых абстракций. Перечисление ключевых абстракций предметной области очень полезно, но в большинстве случаев, кроме самых простых, явно недостаточно. В гл. 7 уже говорилось о том, что очень важно отразить системные решения в объектно-ориентированной форме. Для каждой выявленной нами ключевой абстракции необходимо документально зафиксировать ее входные/выходные характеристики и поведение. Применительно к таймеру это может быть изложено следующим образом:

- | | |
|---------------------------|---|
| * Импортирует (получает) | Сообщение о прохождении интервала времени в 1 с, получаемое извне. |
| * Экспортирует (передает) | Действительное значение времени в секундах с момента включения системы. |
| * Поведение | Передаваемое значение увеличивается на 1 с независимо от состояния обогревателя (активен/пассивен). |

Важно также документировать имеющиеся в отношении каждой абстракции предположения. Для таймера можно записать следующее:

* Предположение

Диапазон значений таймера заведомо достаточен, чтобы избежать переполнения. Например, 32-разрядный таймер может действовать без переполнения более 130 лет.

Внимательный читатель может возразить, что такой способ документирования эквивалентен переписыванию исходных требований с сортировкой их по ключевым абстракциям. Конечно, нет необходимости переписывать все требования, предъявленные к большой системе, поскольку это долгая и утомительная работа. На практике целесообразно документировать таким способом только самые существенные абстракции. Кроме того, этот процесс никогда не происходит сразу, а ведется последовательно по мере выявления ключевых абстракций. Сначала обнаруживаются наиболее важные абстракции в словаре эксперта по данной проблеме, затем делаются обобщения и выявляются экспоненты абстракций. Таким образом, мы постепенно обнаруживаем имеющиеся в требованиях противоречия, уточняем терминологию, устраняем неоднозначности и добиваемся до логики поведения каждой ключевой абстракции.

Создание прототипа

В процессе анализа системы мы добились понимания ключевых абстракций проблемы и главных функций системы. Теперь мы готовы приступить к поиску решений, которые удовлетворяли бы поставленным требованиям. На этой стадии очень важно найти компоненты, которые могут использоваться в программе многократно. Это позволит создавать новые системы методом компоновки из блоков, а не начинать каждый раз заново.

Повторное использование компонент. Активный поиск компонент, которые могут затем использоваться в других задачах, составляет существенный этап проектирования. Этому процессу способствует наличие множества библиотек классов, которые созданы для объектно-ориентированных языков программирования. В частности, в библиотеке классов Smalltalk-80 есть такие компоненты, как `StandardSystemView` (их можно использовать в интерфейсе пользователя). В Smalltalk имеются классы для моделирования измерительных приборов [3] и такие компоненты, как `CircleMeterView` и `BarGaugeWithScaleView`, для моделирования аналоговых приборов (датчиков желаемой и фактической температуры).

Эти классы независимы от предметной области и не могут быть использованы непосредственно. Их требуется дополнить так, чтобы они отражали словарь предметной области. По этой причине мы создадим специализированные классы на основе указанных. Осторожный проектировщик может подумать, что мы уже перешли к этапу реализации. Конечно, начиная создавать прототипы, нужно иметь ясный план, но нельзя избежать ошибок. Главная роль прототипов в нашей задаче состоит в определении того, удовлетворяет ли предложенный интерфейс пользователя поставленным условиям до начала окончательного проектирования. Если интерфейс нас удовлетворяет, мы включим его в перечень абстракций, составляющих систему, и не будем возвращаться к этой проблеме. На основе прототипов мы приходим к соглашению относительно интерфейса пользователя, рассмотрев, возможно, несколько вариантов.

Проектирование класса «переключатель». Рассмотрев перечень ключевых абстракций, мы находим целесообразным сделать обобщение для выключателя отопления и переключателя-индикатора повторного пуска. В обоих случаях мы имеем дело с обычным переключателем. Выключатель отопления можно представить себе как два транспаранта (окна) с надписями «отопление включено» (heat on) и «отопление выключено» (heat off), один из которых должен быть «подсвечен». Для выбора между окнами можно использовать «мышь». При переключении «подсветка» окон меняется местами и передается сообщение в программу об изменении состояния. Выбор «мышью» подсвеченного окна не вызывает никакой реакции.

В библиотеке Smalltalk-80 можно найти класс BooleanView, который пригоден для нашей цели. Этот класс представляет так называемое подключаемое окно, которое может «зажигаться» и «гаситься», и содержит средства индикации на дисплее и реагирования на «мышь». Для образования переключателя нам достаточно два таких переключаемых окна, которые должны находиться в противоположных состояниях.

Класс BooleanView нужно дополнить в соответствии с требованиями задачи и снабдить надписью. Новый экземпляр класса создадим с помощью следующего оператора:

```
aBooleanView <— BooleanView
on: aModel
aspect: #state
label: 'a label'
change: #state
value: true.
```

Это общий прием для всех языков ООП (object-oriented programming), поддерживающих метаклассы: для образования экземпляра класса и его инициализации используется метод соответствующего метакласса.

Класс BooleanView является примером рассмотренного в гл. 4 механизма MVC (model-view-controller). По указанию выше методу создаем новый вид индикации и новый контроллер. Селектор on: указывает на необходимость активизации модели в случае попадания «мыши» в окно (с нажатием). Это позволяет увидеть роль модели: объект видит модель, а контроллер ее изменяет.

Реализация механизма MVC не позволяет объектам быть полностью «видимыми» друг для друга. Контроллер и индикация взаимозависимы и могут обмениваться сообщениями. Каждый из них может направлять сообщения в модель. Модель, напротив, не может прямо направлять сообщения контроллеру и индикатору. Взаимодействие модели и индикации осуществляется косвенно через механизм, изложенный в гл. 3 (каждый объект управляет списком связанных с ним объектов). Поэтому появление нового объекта сопровождается изменением списка зависимостей. Индикация может быть связана только с одной моделью, а модель — с множеством индикаций.

При нажатии на кнопку «мышь» контроллер посылает сообщение state: в модель с указанием булева значения нового состояния. Чтобы модель была полной, необходимо реализовать метод state: так, чтобы выполнить необходимые действия. Класс BooleanView называется *подключаемым окном* потому, что достаточно ввести это в программу, чтобы добиться необходимого эффекта. Такие окна имеют много общего с приемами, используемыми в языках CLU и Ada.

Для образования переключателя необходимо иметь два объекта `BooleanView`. Это лучше всего сделать путем агрегатирования, а не наследования, создав класс `ToggleSwitch` (переключатель), использующий `BooleanView`. Другая причина, по которой использование предпочтительнее, состоит в том, что поведение переключателя является более важным, чем поведение окна. Необходимо, чтобы два окна устанавливались в противоположные состояния при переключениях. Остановимся на таком решении: создаем класс `ToggleSwitch` из суперкласса `Object` путем использования (включения) класса `BooleanView`.

Класс `ToggleSwitch` также необходимо сделать подключаемым. Он не должен быть связан с конкретным содержанием надписи и конкретным способом использования сигнала. Для этого в класс `ToggleSwitch` вводятся четыре селектора: для надписей во включенном и отключенном состоянии и для действий по включению и отключению. Из этого вытекает необходимость введения в этот класс следующих трех переменных:

- * `State` Состояние переключателя вкл./откл.
- * `TrueAction` Блок действий, выполняемых при включении
- * `FalseAction` Блок действий, выполняемых при отключении

Кроме того, нужны еще две переменные, связанные с визуализацией состояния объекта:

- * `trueView` Объект класса `BooleanView`, представляющий включенное состояние
- * `falseView` Объект класса `BooleanView`, представляющий выключенное состояние

Отсюда можно перечислить методы класса `ToggleSwitch`:

```
initialize:            initialValue
trueLabel:            firstLabel
falseLabel:           secondLabel
trueAction:           firstAction
falseAction:          secondAction
```

«Initialize a toggle switch. `InitialValue` is expected to be of the class `Boolean`. `firstLabel` and `secondLabel` are expected to be of the class `String`. `firstAction` and `secondAction` are expected to be of the class `Block`.»

```
state <— initialValue.
trueAction <— firstAction.
falseAction <— secondAction.
trueView <— BooleanView
          on:        self
          aspect:   #state
          label:    firstLabel asText
          change:   #state:
          value:    true.
falseView <— BooleanView
          on:        self
          aspect:   #state
          label:    secondLabel asText
          change:   #state:
          value:    true.
```

Комментарий в примере документирует действие селекторов. Поскольку Smalltalk поддерживает механизм позднего связывания, из текста программы не всегда можно понять, к какому объекту относятся методы. Невнимательный программист может неверно воспользоваться методом. Метод будет реализовываться непредсказуемым образом, пока не обнаружится ошибка при выполнении программы. Динамическое связывание является полезным свойством в определенных случаях, но требует большего внимания от программиста. Для чего мы определили метод инициализации в классе ToggleSwitch, а не воспользовались методом метакласса, как делали раньше? Метод метакласса используется в том случае, когда мы одновременно создаем и инициализируем объект либо при инициализации объекта-переменной.

Определение метода initialize в самом классе позволяет создать объект (с помощью методов new или new:, определенных в классе Behavior), инициализировать его, а позднее инициализировать тот же объект еще раз. Такой подход придает большую гибкость: объект ToggleSwitch можно переинициализировать с другой надписью и другими операциями. При другом подходе придется вводить дополнительный метод для изменения состояния объекта.

Метод state: имеет важное значение для реализации функций данного класса. Поскольку оба объекта BooleanView имеют общую модель, фиксируемые ими события обрабатывают одинаково. Сообщение state: посылается одним из контроллеров trueView или falseView в зависимости от того, какой из них зафиксировал нажатие «мыши». Метод state: должен, следовательно, изменить переменную state, означающую новое состояние объектов. После этого выполняются соответственно методы trueAction или falseAction. Опишем логику следующим образом:

```
state: aBoolean
```

```
«Set the state of the toggle switch. aBoolean is expected to be of the class boolean.»
```

```
state=aBoolean
  ifFalse:
    [state <- aBoolean.
     self changed: #state.
     state
      ifTrue:
        [trueAction value]
      ifFalse:
        [falseAction value]]
```

Осталось неясным только значение сообщения changed: в этом методе. В гл. 3 говорилось уже о том, что в языке Smalltalk используется механизм зависимостей для передачи объектам информации о наличии изменений (update:). В нашем случае сообщение update: направляется объектам trueView и falseView с аргументом #state. Поэтому оба объекта всегда знают о факте переключения состояния, хотя само переключение регистрируется только одним из них. Метод changed: можно было бы заменить следующим двумя операторами:

```
trueView update:    #state.
falseView update:   #state.
```

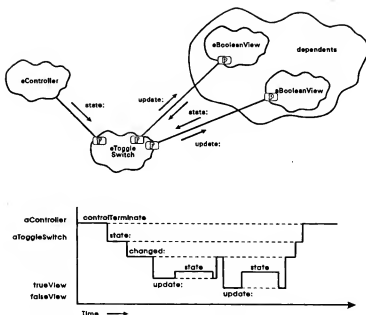


Рис. 8-4. Диаграмма объектов переключателя.

Механизм использования метода `changed:` является более общим и поэтому предпочтительнее. Кроме того, для изменения поведения переключателя в этом случае (метод `changed:`) необходимо модифицировать только метод `state:`. Метод `update:` в классе `BooleanView` должен запросить текущее состояние и «высветить» соответствующее окно. Поскольку объекты-переменные в Smalltalk непосредственно «невидимы» (имеет место ограничение доступа), для определения состояния используется селектор `state:`:

`state`

```
«Return the state of the toggle switch.»
^state
```

Поведение объектов класса `ToggleSwitch` схематически показано на рис. 8-4. На этой схеме видны отношения между переключателем, соответствующим окном и сигналом нажатия «мышь».

Для контроллера переключатель «видим», так как является одним из его полей. Окна переключателя доступны по двум направлениям: через поля `trueView` и `falseView`, а также как связанные компоненты. Все связанные компоненты доступны через итератор соответствующего списка.

На временной диаграмме поясняется процесс управления. При нажатии на «мышь» контроллер фиксирует событие по сообщению `controlTerminate`. Затем посылается сообщение `state:` в модель, а модель формирует сообщение

update: для обоих окон с помощью метода changed:. Каждое из окон отвечает модели сообщением своего нового состояния.

Для полноты определения в переключатель введены методы trueView и falseView, которые являются селекторами, возвращающими значения объектов-переменных. В качестве примера, напомним следующее определение:

trueView

```
«Return the true view of the toggle switch.»
^trueView
```

Ниже показано использование этих селекторов для работы с окнами.

Проектирование класса «выключатель отопления». Уточним структуру класса ToggleSwitch, чтобы образовать класс «выключатель отопления» (HeatSwitch). Поскольку мы имеем дело с разновидностью переключения, используем механизм наследования в соответствии со схемой на рис. 8-5. Класс HeatSwitch образован наследованием от класса ToggleSwitch, который содержит класс BooleanView. Строение класса BooleanView для простоты не показано, поскольку он используется в качестве примитива.

Отметим также, что между классами установлены отношения старшинства. Класс ToggleSwitch является в нашем проекте абстрактным и отмечен уровнем 0 (ноль). Очевидно также, что в Smalltalk метакласс может быть реализован только в виде единственного класса — самого себя. Уровень классов HeatSwitch и BooleanView не показан, так как количество объектов может быть произвольным.

Факт наследования класса HeatSwitch от ToggleSwitch в значительной степени уже определяет его поведение. Нам осталось только уточнить метод инициализации с образованием нужной надписи в окнах этого класса. Сделаем это следующим образом:

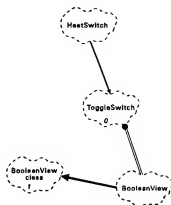


Рис. 8-5. Диаграмма класса «переключатель».

initializeOnAction: firstAction offAction: secondAction

```
«Initialize the heat switch.»
```

```
self
  initialize: false
  trueLabel: 'heat on'
  falseLabel: 'heat off'
  trueAction: firstAction
  falseAction: secondAction
```

Мы не могли здесь воспользоваться переопределением метода инициализации `ToggleSwitch` из-за различного числа параметров.

Проектирование класса «датчик желаемой температуры». По аналогии с переключателем введем еще один универсальный класс. Датчик можно представить себе в виде циферблата, созданного с помощью библиотечного класса `CircleMeterView`. Циферблат — более сложное устройство, чем окно `BooleanView`: пользователь может установить на нем нужное значение некоторого аналогового параметра. Предположим, что проектируемый нами класс `DesiredTemperatureSensor` содержит переменную `value`, означающую выбранное значение температуры, и переменную-объект `theView`, изображающую циферблат. Опшем такой класс с использованием класса `CircleMeterView`.

initialize

```
«Initialize the desired-temperature sensor.»
```

```
! analogGauge !
```

```
super initialize.
analogGauge <— CircleMeterView
  on: self
  aspect: #value
  change: #value
  range: (50 to: 90 by: 5).

theView <— View new.
theView
  addSubView: analogGauge
  in: (0 @ 0 extent: 1 @ 1)
  borderWidth: 0.
value <— 65.0
```

value: aValue

```
«Set the value of the desired-temperature sensor.»
```

```
value = aValue
ifFalse:
  {value <— aValue.
   self changed: #value.
   self changed: #desiredTemperature}
```

Приведенные методы повторяют структуру класса `ToggleSwitch`. Класс `CircleMeterView` является подключаемым. Это, в частности, означает необхо-

димость формирования сообщения в случае действия «мышью» в окне данного класса. Данную функцию решает метод `value:`, действие которого состоит в установлении нового значения желаемой температуры (значение переменной `value`). После этого активизируется метод `changed:`, сообщающий о наличии изменений другим объектам.

Отметим также, что изменение сопровождается посылкой значения `#desired-Temperature` всем объектам, находящимся в отношении зависимости. Внимательный читатель отметит, что эти действия избыточны; посылается также значение `#value`. Эти два посылаемых значения отображают различные виды событий — внутренние и внешние. Для классов `ToggleSwitch` и `Desired-TemperatureSensor` сигналом о наличии изменений (для внутренних объектов `BooleanView` и `CircleMeterView`) является значение `#value`. Кроме того, нужно сообщить о наличии изменений и внешним объектам, которых эти изменения затрагивают, с помощью более информативного сигнала `#desiredTemperature`. Это аналогично использованию метода `Changed:` в классе `ToggleSwitch` для активизации блоков действий `trueAction` и `falseAction` в зависимости от вида событий.

В процессе создания прототипов мы исправили одну ошибку в классе `CircleMeterView`, введя обратный знак для перемещения циферблата в другом направлении. Есть еще одна причина стабильности и устойчивости повторно используемых программных компонент. Обнаружение ошибок на стадии создания прототипов имеет положительный эффект, поскольку в наших интересах обнаружить ошибки и слабые места как можно раньше, до того как появятся проектные решения, которые трудно исправить.

Создание прототипов интерфейса пользователя. На рис. 8-6 приведен снимок с экрана дисплея при создании прототипов на языке Smalltalk интерфейсов оператора и посетителя с помощью классов `ToggleSwitch` и `DesiredTemperatureSensor`, а также двух аналогичных классов `Indicator` и `CurrentTemperatureSensor`. Каждое изображение состоит из комбинаций окон, создаваемых этими классами, объединенных вместе реализацией класса `StandardSystemView`. Интерфейс оператора включает два объекта `ToggleSwitch` и один `Indicator`. Интерфейс посетителя состоит из одного объекта `DesiredTemperatureSensor`, одного `Indicator` и одного `CurrentTemperatureSensor` (который в свою очередь использует класс `BarGangeWithScaleView`).

На этапе создания прототипов выявляется еще один существенный момент: проблема взаимноисключений в интерфейсе пользователя. В частности, мы знаем из требований к системе, что внутреннее событие может управлять переключателем-индикатором, а оператор имеет возможность установить этот переключатель повторно. Если не установить специальных ограничений, оператор может вывести переключатель из строя. Следовательно, при наличии нескольких каналов управления мы должны обеспечить защиту от такого рода вмешательства.

Подтверждается сделанное выше предположение о том, что в процессе проектирования следует постоянно следить за параллельными событиями. Наиболее распространенным приемом управления простыми видами взаимноисключений является использование семафора. Семафор устанавливается в критическом месте программы и позволяет использовать только один из каналов управления. В библиотеке Smalltalk-80 есть класс `Semaphore`, реализующий такую абстракцию, что упрощает модификацию прототипа.

В класс `ToggleSwitch` добавим переменной-объект `theSemaphore`, а в метод инициализации этого класса — еще один оператор:

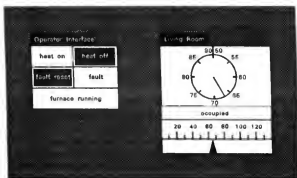


Рис. 8-6. Прототип интерфейса пользователя.

```
theSemaphore <— Semaphore forMutualExclusion.
```

Этот оператор создаст и инициализирует новый объект «семафор» в объекте, представляющем переключатель. Поскольку мы теперь получили активный объект внутри объекта-переключателя, нам нужно быть очень внимательными при работе с семафором при очистке состояния переключателя.

Определим теперь следующий метод:

```
release
```

```
«Release the switch.»
```

```
theSemaphore terminateProcess.
```

```
trueAction <— nil.
```

```
falseAction <— nil.
```

```
super release
```

Нам нужно изменить только один метод state:. В имеющийся код теперь достаточно включить этот метод. Метод state: будет выглядеть так:

```
state: aBoolean
```

```
«Set the state of the toggle switch. aBoolean is expected to be of the class Boolean.»
```

```
theSemaphore critical:
```

```
{state = aBoolean
```

```
ifFalse:
```

```
{state <— aBoolean
```

```
self changed: #state.
```

```
state
```

```
ifTrue:
```

```
{trueAction value}
```

```
ifFalse:
```

```
{falseAction value}]]
```

Теперь в критическую точку программы может вмешаться только один из процессов.



Рис. 8-7. Диаграмма объекта верхнего уровня.

8.2. ПРОЕКТИРОВАНИЕ

Структура объекта

В процессе анализа предметной области и создания прототипа интерфейса мы выделили ключевые абстракции данной области. Мы завершили первый этап OOD — идентификацию классов и объектов верхнего уровня. Теперь необходимо принять проектные решения по структуре найденных абстракций и их взаимоотношениям. Необходимо создать также механизмы связи между абстракциями, чтобы упростить процесс проектирования. Все это составляет логическую модель системы, которую надо представить в виде диаграммы классов и объектов. Выбор языка Smalltalk не требует наличия диаграммы модулей.

В языке Smalltalk базовым элементом модульной декомпозиции является класс и для всех классов, кроме категоризованных (которые являются не столько элементами языка, сколько оболочки Smalltalk), другого способа распределения в пакеты не существует.

Структура объекта верхнего уровня. Структура объекта верхнего уровня для системы домашнего отопления показана на рис. 8-7. На ней показан единственный объект — сама система отопления. Из диаграммы видно, что этот объект имеет несколько статических каналов управления. Базовым классом для этого объекта является HomeHeatingSystem (на рис. это прямо не показано). В отношении данного объекта можно предположить возможность следующих манипуляций:

* Отключение (powerDown).

* Включение (powerUp).

Эти две операции и образуют интерфейсный протокол класса HomeHeatingSystem.

Декомпозиция на верхнем уровне. Объект верхнего уровня имеет внутреннюю структурную иерархию: в класс HomeHeatingSystem должны войти такие объекты, как обогреватель, выключатель, клапаны и т.д. В данной структурной иерархии объекты более низких уровней должны входить в защищенную часть объектов более высокого уровня. Было бы ошибкой сделать все объекты, связанные с ключевыми абстракциями, взаимно видимыми; это только усложнит задачу и не будет отражать действительный уровень абстракций. Гораздо правильнее выбрать из списка ключевых абстракций такие, которые представляют самые крупные, принципиальные фрагменты системы, т.е. абстракции самого верхнего уровня.

Такими фрагментами, по-видимому являются: обогреватель, регулятор, интерфейс оператора и дом. Именно их мы включим в качестве непосредственных компонент в главный объект — систему отопления. Интерфейс посетителей в отличие от интерфейса оператора не включен в состав верхнего

уровня, поскольку он в принципе связан с каждым конкретным помещением. Отметим, что принятое нами решение мало отличается от диаграммы объектов на рис. 8-2. На рис. 8-2 объекты-помещения представлены в виде элементов верхнего уровня, а сейчас мы объединили их в более общую абстракцию — дом. Это решение основано главным образом на том, что имеется по меньшей мере одна общая операция для всех помещений: закрытие всех клапанов подачи воды в случае каких-либо неисправностей в системе.

Детализация объектов верхнего уровня и их связей. Теперь приступим к более детальной проработке объектов верхнего уровня и связей между ними. В гл. 6 рассматривался общепринятый подход к составлению диаграмм объектов, когда все объекты данного уровня абстракции сначала перечисляются, затем попарно анализируются возможные связи между ними и, наконец, определяется протокол связи в виде сообщений между объектами.

Можно предположить, что обогреватель и интерфейс оператора должны быть видимы друг для друга. При включении обогревателя должно послаться сообщение оператору через индикатор состояния обогревателя. Однако в требованиях сказано, что перед включением индикатора должны открываться клапаны подачи воды в комнатах. Это может быть сделано, если обогреватель может запоминать перечень комнат, нуждающихся в подаче тепла, либо запрашивать эту информацию у регулятора. Оба решения имеют недостатки: приходится делать видимыми объекты совершенно различного уровня и структуры. Допустим другой вариант: делаем видимыми обогреватель и дом. Это тоже неверно, поскольку информация о потребности в тепле должна формироваться в комнатах независимо от активности обогревателя. Главная функция обогревателя состоит в генерации тепла, а не в распределении его по комнатам. Последний вариант состоит в наличии связей между домом и интерфейсом оператора. Снова непонятно, что общего между этими двумя объектами.

Приходим к выводу о том, что обогреватель, интерфейс оператора и дом не имеют связи между собой. Все эти объекты связаны двусторонней связью только с регулятором, что соответствует физической структуре системы, показанной на рис. 8-1. В этом нет ничего странного, поскольку процесс проектирования всегда должен отражать логику реальной действительности. Теперь следует выявить операции, которые регулятор может выполнять над объектами, имеющими с ним связь. Подходя к этому вопросу с точки зрения интерфейса объектов, можно предложить следующий перечень:

- * Для обогревателя Включение (activate)
 Выключение (deactivate)
- * Для интерфейса оператора Неисправность (reportFault)
 Состояние обогревателя (reportFurnaceStatus)
- * Для дома Закрыть все клапаны (closeAllWaterValves)
 Закрыть один клапан (closeWaterValve)
 Открыть один клапан (openWaterValve)

Таким образом, образован базовый интерфейс с обогревателем, оператором и домом. Теперь следует рассмотреть ту же проблему с другой стороны и выявить операции тех же трех объектов по отношению к регулятору:

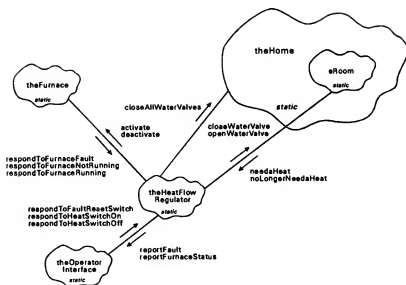


Рис. 8-8. Диаграмма объектов системы отопления.

- От обогревателя
 - Реакция на неисправность (respondToFurnaceFault)
 - Обогреватель не работает (respondToFurnaceNotRunning)
 - Обогреватель работает (respondToFurnaceRunning)
- От оператора
 - Требуется увеличить температуру воздуха (respondToFaultResetSwitch)
 - Требуется увеличить температуру воздуха (respondToHeatSwitchOff)
 - Выключение отопления (respondToHeatSwitchOn)
- От дома
 - Требуется увеличить температуру воздуха (needsHeat)
 - Не требуется увеличить температуру воздуха (noLongerNeedsHeat)

Отметим, что такой протокол строго и логично разделяет все связанные объекты. От обогревателя требуется только выполнять активизацию и деактивизацию и сообщать о неисправности. Интерфейс оператора формирует только соответствующие сообщения оператору. Дом также реализует только самые общие абстракции. Регулятор не знает ничего о таких деталях, как температура в комнатах и есть ли в них кто-нибудь. Потребность в тепле определяется в рамках каждого помещения, где имеется для этого вся информация. Регулятору известно также все, для того чтобы определить, какие клапаны следует открывать.

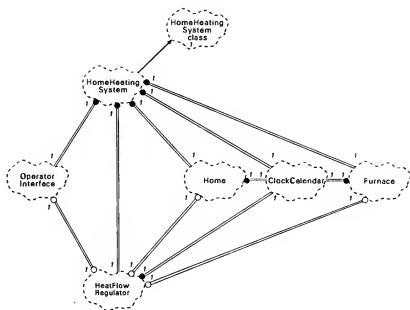


Рис. 8-9. Диаграмма классов верхнего уровня.

Почему на данном уровне абстракции не показаны непосредственно помещения? Дело в том, что абстракция помещения имеет две особенности. Во-первых, с физической точки зрения логично, что комнаты сгруппированы в объект-дом. Во-вторых, имеется операция закрытия сразу всех клапанов во всех помещениях в случае неисправностей в системе отопления. Такую операцию очень удобно реализовывать в одном объекте-доме в виде итератора, который применяется ко всем объектам-помещениям. Поэтому на данном уровне абстракции определен объект-дом, через который осуществляется связь с помещениями.

Все принятые проектные решения отражены на рис. 8-8. Каждый из объектов отмечен определением «статический»: по сути задачи нет смысла динамически создавать или ликвидировать основные объекты системы. Это типично для задач управления процессами. Основные элементы не могут появиться или исчезнуть в системе отопления: они существуют с момента включения системы до ее отключения.

На данном этапе у нас нет оснований судить ни о видах отношений между объектами, ни о природе возможного параллелизма (в системе возможны одновременные события и может потребоваться синхронизация сообщений). Поэтому мы откладываем рассмотрение этих особенностей.

Структура классов

Теперь мы имеем осмысленную структуру абстракций и знаем их связи.

Определены границы каждого объекта и достигнута максимальная степень разделения их структуры. Разумеется, на этом процесс проектирования не заканчивается: определен лишь интерфейс самых важных объектов. Остается детальная проработка каждого объекта, их обобщение и выбор способа представления. Следует отметить, что мы уже частично установили для ряда основных объектов границы соответствующих им классов. Мы имеем теперь все необходимое для того, чтобы создать структуру классов нашей системы, которая является (как было показано в гл. 1) основой любой системы. Не следует стремиться окончательно описать интерфейс всех классов, поскольку этот процесс имеет характер последовательного приближения. В первую очередь мы должны зафиксировать самые важные проектные решения, отложив на время мелкие, несущественные детали, которые будут добавляться по мере функционального насыщения проекта.

Структура классов верхнего уровня. На рис. 8-9 показана структура классов самого верхнего уровня. Отношения использования между классами здесь выражены гораздо сильнее, чем отношения наследования. Система отопления составлена из ряда элементов, таких, как обогреватель или регулятор, но не является разновидностью этих элементов и не эквивалентна сумме составных частей. Поэтому в реализацию класса `HomeHeatingSystem` входят составными частями классы `OperatorInterface`, `HeatFlowRegulator`, `Home` и `Furnace`. Используется также класс `ClockCalendar` (часы/календарь), который будет рассмотрен подробнее. Отметим, что все составляющие систему классы «невидимы» для класса `HomeHeatingSystem`, поскольку сама система является более высоким уровнем абстракции по отношению к составляющим ее классам. На рис. 8-9 отражены также количественные характеристики отношений: существует только по одному объекту каждого используемого класса. В свою очередь каждый используемый класс принадлежит только одной системе отопления.

Из рис. 8-8 было видно, что объекты класса `HeatFlowRegulator` передают сообщения объектам классов `OperatorInterface`, `Home` и `Furnace`, а те в свою очередь могут передавать сообщения объекту класса `HeatFlowRegulator`. Это означает наличие взаимной «видимости» классов `OperatorInterface`, `HeatFlowRegulator`, `Home` и `Furnace`. На рис. 8-9 это отражено в явном виде: двусторонние отношения использования показаны для каждой пары указанных классов.

Например, функции класса `Furnace` имеют доступ к интерфейсу `HeatFlowRegulator`, чтобы иметь возможность передачи сообщений, входящих в интерфейс регулятора (таких, как `respondToFurnaceFault`). Это пример отношений использования в разделе реализации класса. Почему на рис. 8-9 показаны отношения использования для интерфейсной части? Причина этого состоит в том, что объекты этих классов должны инициализироваться экземплярами взаимовидимых классов. В реализации класса `Furnace`, например, имеется следующий оператор:

```
theHeatFlowRegulator respondToFurnaceNotRunning.
```

Этот оператор служит для передачи сообщения `respondToFurnaceNotRunning` объекту `theHeatFlowRegulator`. Из этого вытекает необходимость ис-

пользования классом Furnace интерфейса класса HeatFlowRegulator (сообщение должно быть «видным»). Мы можем поставить вопрос так: каково значение переменной theHeatFlowRegulator? Эта переменная является объектом класса Furnace и должна инициализироваться как объект регулятора. Для этого в классе Furnace имеется следующий метод инициализации:

```
Initialize: aHeatFlowRegulator
```

```
«Initialize the furnace. aHeatFlowRegulator is expected to be of the class HeatFlowRegulator.»
```

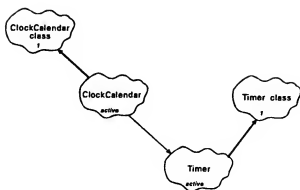
Из примера видно, что класс HeatFlowRegulator должен быть «видным» как для интерфейса, так и для реализации класса Furnace.

Поскольку интерфейс класса Furnace зависит от интерфейса класса HeatFlowRegulator, может показаться, что возникает циклическая завязка при компиляции. В языке Smalltalk такой проблемы не возникает по двум причинам. Во-первых, компиляция в Smalltalk реализует позднее связывание имен с переменными. Это означает, что из факта зависимости интерфейсов Furnace и HeatFlowRegulator не вытекает проблем, поскольку класс конкретного параметра будет определен только в процессе выполнения кода. В строго типизированных языках мы уже не могли бы так легко устранить подобную циклическую зависимость. Почему мы так подробно останавливались на этом вопросе? Потому что решение, связанное с интерфейсной частью, действительно соответствует сути задачи, а не связано с выбранным языком.

Уточнение отношений и структуры классов верхнего уровня. На основе проектных решений, отраженных на рис. 8-9, мы можем приступить к проработке интерфейсов для некоторых классов верхнего уровня. Для класса HomeHeatingSystem можно предложить следующую структуру:

Name:	HomeHeatingSystem
Cardinality:	1
Hierarchy:	
Superclasses:	Object
Metaclass:	HomeHeatingSystem class
Public Interface:	
Operations:	powerDown powerUp
Implementation:	
User:	ClockCalendar Furnace HeatFlowRegulator Home
Fields:	OperatorInterface theFurnace theHeatFlowRegulator theHome theOperatorInterface
Concurrency:	active

Каждый класс в Smalltalk подразумевает наличие метакласса. В процессе проектирования необходимо документировать только основные метаклассы, т.е. те, которые связаны со специфическими для задачи методами. Например, в метаклассе класса HomeHeatingSystem может быть определен метод инициализации объектов этого класса. Такой метакласс может иметь следующую структуру:

Рис. 8-10. Диаграмма класса *Timer*.

Name:	HomeHeatingSystem class
Cardinality:	1
Public Interface:	
Operations:	create

На рис. 8-9 показан класс *ClockCalendar*, хотя в перечне ключевых абстракций он не упоминался. По условиям требовалось наличие только таймера, отсчитывающего интервалы времени в 1 с. Однако упоминались и события с интервалами в минуты (например, время ожидания до перезапуска) и даже дни (период графика посещения комнат). Это означает, что словарь предметной области содержит время в секундах, минутах, часах и днях. Класс *Timer* необходим в проекте, однако есть смысл ввести более абстрактный класс, связанный с обобщением понятия времени. Поэтому введен класс *ClockCalendar*, который является таким же таймером, но позволяет измерять интервалы времени, необходимые для нашей задачи.

На рис. 8-10 изображена структура классов для *ClockCalendar* и *Timer*. Заметим, что класс *ClockCalendar* унаследован от класса *Timer* и что оба класса имеют свои метаклассы. Метаклассы нужны для инициализации объектов; например, при создании объекта часы/календарь инициализируются параметры неделя, час, минута и секунда, чтобы синхронизировать работу системы с реальным временем. Время должно быть единым для всей системы, поэтому интервал в секундах регистрируется в переменной класса, чтобы все объекты класса *Timer* и его подклассов использовали общее показание времени. Чтобы инициализировать такую переменную класса, необходимо иметь соответствующие методы метакласса.

У внимательного читателя может возникнуть вопрос о необходимости разделения классов *Timer* и *ClockCalendar*. Это разделение сделано для того, чтобы расширить возможность повторного использования данных классов в других задачах. Класс *Timer* позволяет только регистрировать интервалы времени в 1 с. Класс *ClockCalendar*, напротив, связан с особенностями предметной области и реализует события более высокого уровня.

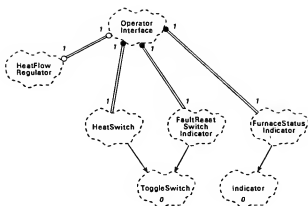


Рис. 8-11. Диаграмма классов интерфейса оператора.

Из рис. 8-10 видно, что объекты класса `Timer` (а следовательно, и класса `ClockCalendar`) имеют активный канал управления. В системах, связанных с реальным временем, как и в случае с системой отопления, состояние системы изменяется в соответствии с ходом времени. Поэтому необходимо регистрировать временные события и лучше всего это сделать с помощью простейшего класса `Timer`. Теперь мы отложим на время рассмотрение процесса регистрации времени и обратим внимание на организацию процессов системы в целом.

На рис. 8-11 показана принятая нами в проекте структура классов интерфейса оператора. На этапе создания прототипов мы определили, что этот интерфейс содержит выключатель отопления, переключатель-индикатор повторного пуска и индикатор состояния обогревателя. Следовательно, класс `OperatorInterface` должен использовать в своей реализации классы составляющих его объектов. Интерфейс оператора не является простой суммой этих объектов, поэтому в данном случае (как и для `HomeHeatingSystem`) уместнее отношения использования, а не наследования. Использование относится только к реализации, но не к интерфейсу, так как `HeatSwitch`, `FaultResetSwitchIndicator` и `FurnaceStatusIndicator` являются защищенными объектами в `OperatorInterface`.

Класс `HeatSwitch` унаследован от класса `ToggleSwitch`. Из рис. 8-11 видно, что класс `FaultResetSwitchIndicator` также унаследован из того же абстрактного класса. Мы исходим из того, что объект переключатель-индикатор больше похож на переключатель, чем на обычный индикатор.

Можно ли здесь воспользоваться множественным наследованием? В частности, можно ли образовать класс `FaultResetSwitchIndicator` из классов `ToggleSwitch` и `Indicator`? Ответ будет отрицательным, поскольку переключатель-индикатор не соответствует поведению комбинации переключателя и индикатора. Это простой переключатель, который может быть установлен пользователем только в одно из положений, а программой в другое. Это поведение является разновидностью поведения класса `ToggleSwitch`.

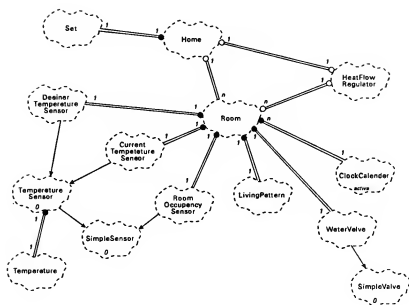


Рис. 8-12. Диаграмма классов *Home* (Дом).

Структура класса *Home* показана на рис. 8-12. С логической точки зрения дом — это совокупность комнат. Следует ли из этого, что класс *Home* должен быть подклассом предопределенного класса *Set*? Ответ вновь отрицательный. Дом действительно состоит из помещений, но его поведение не является поведением обычного множества (*Set*), которое включает операции включения, исключения, объединения и пересечения. Мы ранее уже упоминали, что для дома в целом определена одна главная операция *closeAllWaterValue* и две операции в отношении отдельных помещений — *openWaterValue* и *closeWaterValue*. Операции над множествами здесь неуместны.

Можно возразить, что дом имеет черты множества и должен рассматриваться как подкласс от класса *Set*. Проектировщик всегда готов к таким возражениям. Конечно, использование предопределенного класса ускоряет работу над проектом и такой подход часто используется для создания прототипа. Но если мы хотим создать продукт, который затем будет использоваться и развиваться в течение долгого времени, то следует стремиться отражать суть абстракции. Это требует дополнительных затрат в процессе проектирования, но окупается в процессе эксплуатации продукта.

Для класса *Home* мы установили, что в его реализации можно воспользоваться классом *Set* (для отражения перечня помещений), а в интерфейсной части следует использовать класс *Room* (помещение). Вынесение класса *Room* в интерфейс объясняется тем, что регулятор тепла должен иметь доступ ко всем помещениям в доме. В количественном отношении дом может

состоять из n помещений, но каждое из помещений принадлежит одному дому. Оставшаяся часть диаграммы не требует пояснений. Приведенная структура соответствует иерархии, показанной на рис. 8-3. Класс Room использует (включает в себя) классы DesiredTemperatureSensor, CurrentTemperatureSensor, LivingPattern, WaterValue и RoomOccupancySensor. Количественные соотношения везде взаимоднозначны. Все комнаты связаны с общим регулятором тепла, который может посылать сообщения в любое помещение.

На рис. 8-12 представлено несколько новых классов. При разработке класса для комнаты мы нашли возможность обобщения. Таким образом был введен абстрактный класс датчика (Temperature Sensor) как для желаемой, так и для текущей температуры, измеряющей величины по Фаренгейту. Другим еще более общим абстрактным классом является SimpleSensor, который соответствует получаемому от внешнего устройства значению, без уточнения его смысла и способа получения. Для отражения особенностей измерения температуры по Фаренгейту (диапазон и точность) используется класс Temperature. Поэтому в абстрактном классе TemperatureSensor (унаследованном от класса SimpleSensor) введено значение класса Temperature.

Следует обратить внимание на то, что не все принятые проектные решения могут быть реализованы на языке Smalltalk. В частности, допустив, что температурный датчик — это простой датчик, измеряющий температуру по Фаренгейту, мы изменили структуру класса. Разумно предположить, что диапазон измерения лежит в пределах 0-300° F с дискретностью 0,1° F. Поскольку в Smalltalk используется только позднее связывание объектов с именами, мы не имеем средств, позволяющих непосредственно реализовать это проектное решение. В языках со строгой типизацией, таких, как Object Pascal и Ada, эти решения было бы можно реализовать на этапе компиляции. Другим примером затруднений, связанных с языком Smalltalk, является невозможность отражения некоторых количественных отношений. Из рис. 8-12 видно, что TemperatureSensor, SimpleSensor и SimpleValue являются абстрактными классами. Но Smalltalk не позволяет легко ограничить число объектов данного класса. Нам пришлось бы использовать сомнительный прием образования метода new в метаклассах, что значительно усложнило бы задачу. Для реализации принятых проектных решений на Smalltalk придется пересматривать код или использовать другие средства анализа. В нашем случае мы можем найти способы обхода ограничений, но в больших проектах желательно использовать языки, непосредственно поддерживающие подобные проектные решения. Об ограниченных возможностях Smalltalk свидетельствует проблема «видимости» классов. Все классы Smalltalk глобальные. Из рис. 8-12 следует, что только класс Room использует в своей реализации класс LivingPattern. Это решение принципиальное, так как в больших системах с огромным числом классов невозможно сконцентрировать внимание на отдельных деталях поведения без ограничения видимости. К сожалению, в Smalltalk нет механизма таких ограничений.

Если используемый язык программирования не позволяет реализовывать некоторые проектные решения, то зачем мы их документируем? Смысл этой работы в том, чтобы лучше разобраться в логике проекта, а главное оставить «следы» наших рассуждений для тех, кто будет сопровождать и развивать проект в дальнейшем. Программа на ассемблере не позволяет быстро войти в логику кода. Означает ли это, что мы должны оставить комментарии к проекту только в его коде? Конечно нет, поэтому мы оставляем в документации и те решения, которые в коде не могут найти отражения.

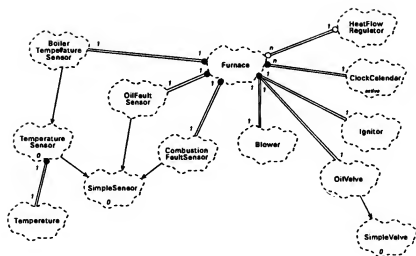


Рис. 8-13. Диаграмма классов для обогревателя.

На рис. 8-13 приведена диаграмма классов обогревателя, аналогичная структуре, показанной на рисунке 8-12. Мы видим, что обогреватель состоит из датчика температуры воды, датчика подачи топлива, датчика сбоев горения, котла, клапана топлива и воспламенителя. Класс *Furnace* включает соответствующие этим объектам классы.

Класс *OilValve* обобщен до уровня *SimpleValve* (простого клапана), который на рис. 8-12 показан в качестве суперкласса *WaterValve*. Клапаны для воды и топлива управляют подачей разных жидкостей, но и тот и другой могут только открываться и закрываться.

Таким образом, основа системы создана: мы определили ключевые абстракции, их отношения и границы. Теперь следует выделить главные функциональные фрагменты системы, описать их интерфейс и осуществлять реализацию. Но прежде чем приступить к такой детальной проработке, мы должны установить архитектуру процессов системы, учитывая результаты этапа создания прототипов.

Архитектура процессов

В гл. 2 показано, что задачи, решаемые системой, обычно можно разложить на достаточно независимые подзадачи. До сих пор мы занимались разработкой структуры классов, отражающих логику ключевых абстракций. Теперь на основе полученного статического каркаса мы можем начать рассмотрение потоков управления в разрабатываемой системе.

Каналы управления. Определим, где в нашей системе находятся каналы управления. Один активный процесс мы уже называли в связи с таймером (а следовательно, и с часами/календарем). Анализируя систему, мы обнаружим множество асинхронных событий, которые влияют на ее поведение и могут быть одновременными. Например, одновременно в двух комнатах мо-

жет быть установлено новое значение желаемой температуры, кто-то покидает третью комнату, а в четвертой комнате упала температура (открыто окно). Из этого следует, что датчики системы тоже представляют активные каналы управления. Большую часть времени они находятся в пассивном ожидании событий. Но теоретически эти события могут происходить одновременно во всех датчиках. Попытаемся проследить за последствиями двух таких событий в системе и сделать так, чтобы архитектура процессов защищала общие данные и сохраняла возможность использования преимуществ параллелизма.

Объекты класса `ClockCalendar` определяют интервалы времени, используемые другими объектами системы. Возвращаясь к исходным требованиям, вспоминая, что только два класса объектов реагируют на временные события — обогреватель и недельный график. Обогреватель использует время для двух целей: вентилятор может останавливаться через 5 с после закрытия клапана топлива, а повторное включение обогревателя не может выполняться ранее чем через 5 мин после остановки. Реализация такого поведения требует регистрации соответствующих событий и лучше всего это делать в самом объекте-печке. Аналогично недельный график посещений должен учитывать ход времени и определять ожидаемое время появления в помещениях людей в пределах 30 мин. Каждые полчаса отмечаются изменения в графике посещений.

Возникает вопрос: как отмечать в этих объектах течение времени? Очевидный ответ состоит в том, что часы/календарь должны посылать в эти объекты соответствующую информацию. Но откуда часы/календарь будут знать, кому в данный момент нужна информация? Следует вновь воспользоваться механизмом зависимостей `Smalltalk`. Достоинство этого механизма в том, что часы/календарь отделяются от классов `Furnace` и `LivingPattern`. Класс `ClockCalendar` даже не должен быть «видим» для этих двух классов, а это означает большие возможности повторного использования классов и также упрощает сам класс `ClockCalendar`, делая его более абстрактным.

Данное проектное решение отражено на рис. 8-14. Обогреватель и график делаются зависимыми по отношению к объекту класса `ClockCalendar` (который является глобальным). По мере прохождения времени часы/календарь посылают сообщения `change`, а затем `update`: каждому из зависимых объектов; при этом часы/календарь не обязаны «знать», сколько имеется зависимых объектов от одного класса `ClockCalendar`, так как они могут управлять любым количеством графиков, поэтому часы/календарь не изменяются в зависимости от числа комнат, обслуживаемых системой.

Отметим наличие доступа к часам/календарю от зависимых объектов (через итератор по всем зависимым объектам), а зависимые объекты доступны для часов/календаря непосредственно.

Объект часы/календарь должен быть доступен для таких объектов, как обогреватель, поскольку для них требуется отсчет времени. Доступ к часам/календарю осуществляется путем включения класса `ClockCalendar`; логически оба объекта оказываются в общей зоне «видимости».

Так как мы хотим, чтобы время для всех связанных объектов было единым, мы должны образовать класс `Timer` и его метакласс, позволяющие хранить нужные значения в переменных класса, а не в переменных объектов. `Timer` в нашей задаче моделирует активный процесс, изменяющий каждую секунду значение единственной переменной. Для метакласса `Timer` структура может быть следующей:

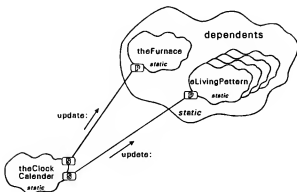


Рис. 8-14. Диаграмма объекта часы/календарь (Clock/Calendar).

Name:	Timer class
Cardinarchy:	1
Public Interface:	
Operations:	elapsedSeconds elapsedSeconds initialize release
Implementation:	
User:	Semaphore
Fields:	elapsedSeconds TheSemaphore TimerProcess
Concurrency:	active

Методы initialize и release для этого метакласса будут следующими:

initialize

```

«Reset elapsedSeconds, then start the process to update elapsedSeconds every second.»
TheSemaphore <— Semaphore forMutualExclusion.
ElapsedSeconds <— 0.
TimerProcess <— [[Delay forSeconds: 1) wait.
                  self elapsedSeconds: self elapsedSeconds+1
                  true] whileTrue] newProcess.
TimerProcess resume

```

release

```

«Stop the process to update elapsedSeconds every second.»
TheSemaphore terminateProcess.
TimerProcess Terminate

```

В методе инициализации создаются новый семафор и процесс таймера. Этот процесс, находясь в состоянии активизации, возбуждается каждую секунду и увеличивает на единицу переменную ElapsedSeconds.

Теперь запишем следующее определение:

elapsedSeconds: anInteger

«Set the number of seconds that have elapsed since the object was initialized. anInteger is expected to be of the class Integer.»

TheSemaphore critical:

```
[ElapsedSeconds <— anInteger.
 self changed: #elapsedSeconds]
```

Уже говорилось, что класс ClockCalendar является подклассом класса Timer, ориентированным на особенности конкретной задачи. В классе ClockCalendar добавляются методы пересчета секунд в минуты, часы, сутки и дни недели. Чтобы непосредственно наследовать от класса Timer его поведение, в класс Timer вводятся переменные класса DayOfWeek, Hour, Minute, Second. Эти переменные должны быть инициализированы в метаклассе ClockCalendar следующим образом:

initializeDayOfWeek: aDay hour: anHour minute: aMinute

..initialize the clock/calendar. aDay is expected to be of the class Integer (range 1 to 7), anHour is expected to be of the class Integer (range 0 to 23), and aMinute is expected to be of the class Integer (range 0 to 59).»

```
DayOfWeek <— aDay.
Hour <— anHour.
Minute <— aMinute.
Seconds <— 0.
super initialize
```

В требованиях к системе говорится о двух временных событиях: 5 с (для выключения вентилятора) и 30 мин (для графика посещений). Задержка в 5 мин для перезапуска обогревателя будет реализована иначе, поскольку в этом случае нужно знать время последней остановки обогревателя. Учитывая это, изменим содержание метода elapsedSeconds, чтобы он отвечал условиям двух указанных выше событий. Метод elapsedSeconds в метаклассе ClockCalendar будет следующим:

elapsedSeconds: anInteger

«Set the number of seconds that have elapsed since the object was initialized, and update the day of the week, hour, and minute. anInteger is expected to be of the class Integer.»

TheSemaphore critical:

```
[ElapsedSeconds <— anInteger.
 self changed: #elapsedSeconds.
 (Seconds rem: 5) = 0
 ifTrue:
    [self changed: #fiveSecondEvent].
 Seconds <— Seconds + 1.
 Seconds = 60
 ifTrue:
```

```

[Seconds <— 0.
  (Minute rem: 30) = 0
    ifTrue:
      [self changed: #thirtyMinuteEvent].
Minute <— Minute + 1.
Minute = 60
  ifTrue:
    [Minute <— 0.
     Hour <— Hour + 1.
     Hour = 24
       ifTrue:
         [Hour <— 0.
          DayOfWeek <— DayOfWeek + 1.
          DayOfWeek = 8
            ifTrue:
              [DayOfWeek <— 1]]]]]

```

Поскольку мы сделали таймер активным объектом, мы должны реализовать класс *LivingPattern* (распорядок дня) в виде блокированного класса, чтобы избежать столкновения различных процессов (процессов таймера и процессов в помещении). Необходимо принять меры предосторожности, так как объекты этого класса имеют несколько каналов управления. Использование семантики блокирования (внутренний семафор, аналогичный классу *ToggleSwitch*) удовлетворяет этим требованиям.

Такой же параллелизм возможен и для обогревателя. Показанный подход к решению проблемы одновременных событий является хорошим примером разработки архитектуры процессов. Чаще всего поиск вероятного параллелизма ведется со стороны внешних событий. Если такие (параллельные) события обнаруживаются, то к соответствующим объектам следует добавить методы их разрешения.

Асинхронные события. Мы установили, что асинхронные события возможны в датчиках температуры (желаемой и фактической) и в датчиках присутствия людей. Следовательно, соответствующие классы должны быть активными. Было показано, как класс *DesiredTemperatureSensor* сообщает другим зависимым объектам о факте изменения пользователем значения желаемой температуры в той или другой комнате (через механизм зависимостей языка Smalltalk). Этот прием можно использовать для других датчиков. Объект-комната может регистрировать состояние всех своих датчиков, а датчики возвращать сообщения о наличии изменений *update*: в объект-комнату. Как и в случае с классом *ClockCalendar*, это сильно связывает объекты. Логично предположить, что в классе *Room* также следует предусмотреть блокировку. Сам класс *Room* является активным, поскольку в него входят активные объекты; то же самое относится и к классам *OperatorInterface* и *Furnace*. Классы *ToggleSwitch*, *BoilerTemperatureSensor*, *OilFaultSensor* и *CombustionFaultSensor* являются активными, так как они должны реагировать на асинхронные (потенциально одновременные) события. По этой причине активны и классы *OperatorInterface* и *Furnace*.

Следовательно, мы можем внести соответствующие изменения в диаграммы классов и объектов, отражающие архитектуру процессов в системе. На рис. 8-15 показана диаграмма объектов системы отопления с учетом свойств параллельности для основных объектов. Из этой диаграммы видно три потенциальных канала управления: обогреватель, интерфейс оператора и

комнаты. Регулятор подачи тепла не имеет собственных активных процессов, но требует наличия блокировки. Объект-дом должен быть последовательным, поскольку лишь один процесс может влиять на его состояние в каждый момент времени (а состояние комнат может быть подвержено воздействию нескольких разных процессов). Отметим, что на рис. 8-15 видно, как все сообщения в системе могут блокироваться.

Читатель может задать вопрос об уместности анализа архитектуры процессов на данном этапе. В гл. 2 упоминалось о задачах, связанных с множественностью каналов управления (такой задачей является и система отопления). В этих системах нельзя заранее предвидеть последовательность событий, т.е. мы теряем способность предсказать поток управления. Поэтому очень важно спроектировать архитектуру процессов, чтобы избежать потенциальных столкновений, вызываемых наличием нескольких каналов управления. В гл. 3 и 4 было показано, что существует хорошая методология перехода от объектно-ориентированного проектирования к архитектуре процессов, которая позволяет рассматривать каждый объект как узел управления. Поскольку объекты обладают внутренним защищенным состоянием, это состояние может быть сохранено в условиях множества каналов управления.

По отношению к любому проекту мы обязаны ответить на очень важный вопрос о том, как изменяется поведение системы в разных условиях. Частично мы это уже сделали, рассмотрев воздействие на систему внешних событий. Есть другой интересный вопрос: что делает система в отсутствие внешних событий? В этом случае все процессы либо находятся в состоянии ожидания, либо заблокированы, а система пассивна. Действительно, в отсутствие внешних событий таймер каждую секунду возбуждается, а остальное время ожидает прохождения интервала времени. С этой точки зрения мы получили стабильную систему с поведением, отражающим структуру классов и архитектуру процессов. С помощью диаграмм объектов мы выполнили моделирование отдельных динамических аспектов поведения. И теперь есть все необходимо, чтобы завершить полную реализацию системы. Мы создадим несколько исполняемых систем по мере заполнения проекта функциональными свойствами, чтобы последовательно оценить принимаемые проектные решения и обеспечить обратную связь с пользователями системы (с заказчиком).

8.3. РЕАЛИЗАЦИЯ

Реализация интерфейса пользователя

Принятые проектные решения, отраженные на диаграммах классов и объектов (рис. 8-11 и рис. 8-15), позволяют перейти к проработке интерфейса класса `OperatorInterface`, а затем к реализации его методической части. Наше представление этого класса можно отразить в следующей структуре:

Name:	<code>OperatorInterface</code>
Cardinality:	1
Hierarchy:	
Superclasses:	<code>Object</code>
Public Interface:	
Uses:	<code>HeatFlowRegulator</code>
Operations:	<code>heatStatus</code> <code>initialize</code> <code>release</code>

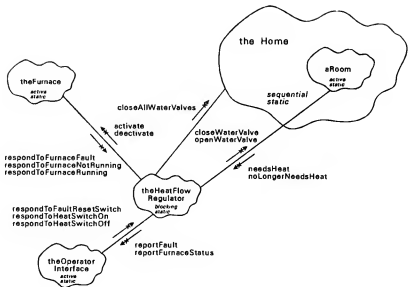


Рис. 8-15. Модифицированная диаграмма объектов системы отопления.

	reportFault
	reportFurnaceStatus:
	Implementation:
Uses:	FaultResetSwitchIndicator
	FurnaceStatusIndicator
	HeatSwitch
Fields:	theFaultResetSwitchIndicator
	theFurnaceStatusIndicator
	theHeatFlowRegulator
	theHeatSwitch
	theView
Concurrency:	active

Иерархическая структура интерфейса оператора показана на рис. 8-16. Видно, что соответствующий объект состоит из двух переключателей (switch) и одного индикатора (indicator). Переключатели и индикатор являются полями объекта. Интерфейс класса `OperatorInterface` имеет доступ к регулятору, а следовательно, и для полей класса регулятор также доступен.

Рассмотрим возможные потоки управления между этими объектами. Регулятор может обращаться к интерфейсу оператора сообщением `ReportFault`. Интерфейс оператора может реагировать на это сообщение передачей сообщения `state`: переключателю-индикатору. Эти сообщения могут временно блокироваться в соответствии с изложенным выше подходом к параллелизму.

Выключатель отопления также может посылать сообщение `respondHeatSwitchOn` регулятору. Реализация класса `ToggleSwitch`, изложенная выше, предусматривает два блока инициализации, выполняемые при включе-

нии и отключении переключателя. Соответствующие блоки должны быть включены и в интерфейс оператора для связи с выключателем. Аналогичный блок необходим и для переключателя-индикатора повторного запуска.

Для простоты на рис. 8-16 не показаны сообщения, пересылаемые от интерфейса оператора составляющим его полям при инициализации. Полное описание инициализации выглядит следующим образом:

initialize: anHeatFlowRegulator

«Create the operator interface for the given heat flow regulator. aHeatFlowRegulator is expected to be of the class HeatFlowRegulator.»

theHeatFlowRegulator <— aHeatFlowRegulator.

theHeatSwitch <— HeatSwitch new.

theHeatSwitch

initializeOnAction:

[theHeatFlowRegulator respondToHeatSwitchOn]

offAction:

[theHeatFlowRegulator respondToHeatSwitchOff].

theFaultResetSwitchIndicator <— FaultResetSwitchIndicator new.

theFaultResetSwitchIndicator

initializeOnAction:

[theHeatFlowRegulator respondToFaultResetIndicatorOn].

theFurnaceStatusIndicator <— FurnaceStatusIndicator new.

theFurnaceStatusIndicator initialize.

theView <— RestrictedSystemView

model: nil

label: 'Operator Interface'

minimumSize: 150 @ 75.

theView borderWidth: 1.

theView

addSubview: theHeatSwitch trueView

in: (0 @ 0 extent: 1 / 2 @ (1 / 3))

borderWidth: 1.

theView

addSubview: theHeatSwitch falseView

in: (1 / 2 @ 0 extent: 1 / 2 @ (1 / 3))

borderWidth: 1.

theView

addSubview: theFaultResetSwitchIndicator trueView

in: (0 @ (1 / 3) extent: 1 / 2 @ (1 / 3))

borderWidth: 1.

theView

addSubview: theFaultResetSwitchIndicator falseView

in: (1 / 2 @ (1 / 3) extent: 1 / 2 @ (1 / 3))

borderWidth: 1.

theView

addSubview: theFurnaceStatusIndicator IndicatorView

in: (0 @ (2 / 3) extent: 1 @ (1 / 3))

borderWidth: 1.

theView controller openDisplayCentered: 100 @ 125

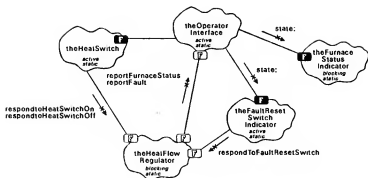


Рис. 8-16. Диаграмма объектов интерфейса оператора.

Отметим наличие в этом методе аргумента в виде объекта класса `HeatFlowRegulator`. Этот аргумент является частью структуры состояния интерфейса оператора и используется при инициализации блоков, передаваемых в выключатель и в переключатель-индикатор. В частности, при включении отопления выполняется следующий блок:

```
[theHeatFlowRegulatorrespondToHeatSwitchOn]
```

Остальной код служит для визуализации интерфейса оператора. Поле `theView` инициализируется объектом класса `RestrictedSystemView`. Этот класс не был показан в структуре класса `OperatorInterface` и на рис. 8-11 для простоты, но в реализации класса он используется. Класс `RestrictedSystemView` образован из предопределенного класса `StandardtSystemView`, но не позволяет пользователю переименовывать окно изображения.

К объекту `theView` добавлены дополнительные окна. Селекторы `TrueView` и `falseView` в классе `ToggleSwitch` необходимы для прямого доступа к этим двум окнам. Метод завершается активизацией контроллера изображения, позволяющего отобразить окно на экране дисплея.

Реализация класса-помещения

Для класса `Room` структура может быть следующей:

Name:	Room
Cardinarchy:	n
Hierarchy:	
Public Interface:	Superclasses: Object
Uses:	HeatFlowRegulator
Operations:	closeWaterValve
	initialize: location: heatFlowRegulator:
	name
	openWaterValve
	release
Implementation:	

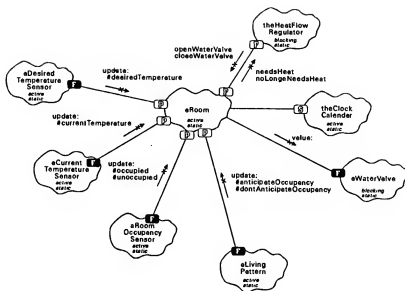


Рис. 8-17. Диаграмма объектов для помещения.

Uses:	ClockCalendar CurrentTemperatureSensor DesiredTemperatureSensor LivingPattern WaterValve
Fields:	currentState name theCurrentTemperatureSensor theDesiredTemperatureSensor theHeatFlowRegulator theLivingPattern theRoomOccupancySensor theSemaphore theView theWaterValve
Operations:	expectAndHeating: current: desired: occupied: expected: expectAndNoHeating: current: desired: occupied: expected: occupiedAndHeating: current: desired: occupied: expected: occupiedAndNoHeating: current: desired: occupied: expected: setInitializeState startHeating stopHeating unoccupiedAndHeating: current: desired: occupied: expected: unoccupiedAndNoHeating: current: desired: occupied: expected: update: active
Concurrency:	

Интерфейс этого класса состоит из операторов, показанных на диаграмме объектов рис. 8-15 и группы операций инициализации, устранения и изображения. Кроме уже упоминавшихся составляющих класса `Room` в него включен ряд обособленных операций, составляющих основу поведения объектов этого класса.

На диаграмме объектов (рис. 8-17) показано взаимодействие объекта-помещения с другими объектами. Из диаграммы видно, что помещение и часы/календарь находятся в общей зоне видимости. Операции взаимоотношения на рисунке не показаны. При инициализации объекта-помещения последний регистрирует расписание дня по часам/календарю и больше к этому объекту не обращается. Некоторые объекты входят в состав объекта-помещения в качестве полей: датчик желаемой температуры, датчик текущей температуры, датчик присутствия людей, расписание дня и клапан подачи воды. Все эти объекты связаны с помещением механизмом зависимости языка `Smalltalk`. Одной из операций инициализации объекта-помещения и его полей является установление отношений зависимостей с датчиками и расписанием дня. Для данной группы объектов рассмотрим поток управления между комнатой и регулятором. Регулятор тепла посылает в комнату сообщение на открытие или закрытие клапана подачи воды. Комната в ответ на это возвращает сообщение `value:` с аргументом, который означает состояние клапана. Аргумент является видимым, как одно из полей класса `Room`. Каждый из этих методов может блокироваться согласно логике обработки параллельных событий.

На основании каких данных помещение посылает сигналы регулятору на возобновление или прекращение подачи тепла? В объекте-помещении есть все необходимые данные для формирования таких сообщений-сигналов. Действительно, объект-помещение содержит механизм контроля состояния, управляемый событиями, формируемыми в пределах объекта-помещения: датчики температуры, наличие людей и расписание дня. Из рис. 8-17 видно, что каждый из объектов, входящих в структуру `Room`, посылает сообщение `update:` с символьным параметром. Символьный параметр характеризует какое-либо внешнее событие, на которое помещение должно отреагировать. Поскольку система пассивна, порядок прохождения событий является существенным и механизм контроля состояния объекта класса `Room` должен упорядочить поведение системы.

Исходные требования говорят о том, что каждое помещение может быть либо свободно, либо свободно, но ожидает посетителей, либо занято людьми. Клапан подачи воды в комнату может быть открыт (комната обогревается) или закрыт (комната не обогревается). В результате можно выделить шесть возможных состояний объекта-помещения:

- * Помещение свободно и не обогревается.
- * Помещение свободно и обогревается.
- * Ожидание посещения без обогрева.
- * Ожидание посещения с обогревом.
- * Присутствие людей без обогрева.
- * Присутствие людей с обогревом.

При создании объекта-помещения необходимо установить его начальное состояние. При включении системы отопления помещения могут быть как заняты, так и свободны, а некоторые уже нуждаются в обогреве. Для инициализации исходного состояния помещения введен отдельный метод `setInitialState`.

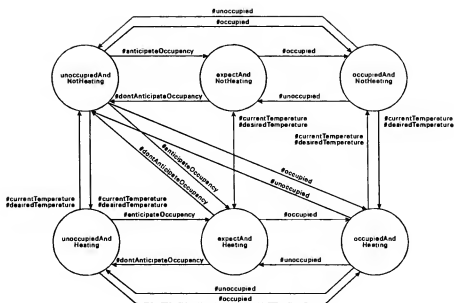


Рис. 8-18. Диаграмма перехода состояния для помещения.

На рис. 8-18 приведена диаграмма переходов состояний для класса Room. Эта диаграмма не устанавливает определенного исходного и конечного состояния, поскольку исходное состояние определяется динамически, а функционирование системы не прекращается до ее полного отключения.

Преобразование такой диаграммы переходов в код Smalltalk не представляет особых трудностей. В качестве сигналов для класса Room можно использовать метод update:, который реализуется через механизм зависимостей для всех объектов, входящих в состав помещения. Одной из возможностей реализации метода update: является использование составного оператора if, который в зависимости от состояния объекта выбирает соответствующие методы реакции. Реакцией на какое-либо событие являются изменение состояния и формирование сообщений с помощью упоминавшихся выше отдельных методов. Теперь мы можем реализовать рассматриваемый метод:

update: aMessage

«Update the state of the room. Possible states are #occupiedAndHeating, #occupiedAndNoHeating, #expectAndHeating, #expectAndNoHeating, #unoccupiedAndHeating, #unoccupiedAndNoHeating. Possible events are #occupied, #unoccupied, #currentTemperature, #desiredTemperature, #anticipateOccupancy, and #dontAnticipateOccupancy. Messages may be sent to request heat or to stop heat.»

| current desired occupied expected |

theSemaphore critical:

[aMessage = #value

```

ifFalse:
    [current <= theCurrentTemperatureSensor value.
     desired <= theDesiredTemperatureSensor value.
     occupied <= theRoomOccupiedSensor value.
     expected <= theLivingPattern occupancyExpected.
     currentState = #occupiedAndHeating ifTrue:
        [self
         occupiedAndHeating: aMessage
         current: current
         desired: desired
         occupied: occupied
         expected: expected] ifFalse:
        [currentState = #occupiedAndNoHeating ifTrue:
         [self
          occupiedAndNoHeating: aMessage
          current: current
          desired: desired
          occupied: occupied
          expected: expected] ifFalse:
          [currentState = #expectAndHeating ifTrue:
           [self
            expectAndHeating: aMessage
            current: current
            desired: desired
            occupied: occupied
            expected: expected] ifFalse:
            [currentState = #expectAndNoHeating ifTrue:
             [self
              expectAndNoHeating: aMessage
              current: current
              desired: desired
              occupied: occupied
              expected: expected] ifFalse:
              currentState = #unoccupiedAndHeating ifTrue:
               [self
                unoccupiedAndHeating: aMessage
                current: current
                desired: desired
                occupied: occupied
                expected: expected] ifFalse:
               [currentState = #unoccupiedAndNoHeating ifTrue:
                [self
                 unoccupiedAndNoHeating: aMessage
                 current: current
                 desired: desired
                 occupied: occupied
                 expected: expected]]]]]]]]]]

```

В качестве альтернативной реализации можно воспользоваться операцией `perform` для выбора действий в зависимости от текущего состояния. Недостатком такого подхода является необходимость распаковки массива аргументов, используемого операцией `perform`. В каждом отдельном методе реализуется аналогичный селектор по символу, обозначающему событие, текущее состояние, желаемую температуру, наличие людей, ожидание людей. Исходя из этих аргументов, можно реализовать отдельные методы, реализующие переходы согласно диаграмме на рис. 8-18. Для примера рассмотрим метод `occupiedAndNoHeating:`. Изменение состояния `occupiedAndNotHeating` может быть вызвано тремя событиями (рис. 8-18):

- * Помещение освободилось.
- * Изменилась желаемая температура.
- * Изменилась фактическая температура.

В первом случае осуществляется переход в состояние `expectAndNoHeating` или `unoccupiedAndNotHeating` (в зависимости от состояния ожидания, фиксируемого расписанием дня). В двух других случаях (если фактическая температура стала ниже желаемой более чем на два градуса) потребуется подача тепла. Зафиксируем эту логику в следующем обособленном методе:

```
occupiedAndNoHeating: aMessage
current: current
desired: desired
occupied: occupied
expected: expected
```

«Respond to a state change. desired and occupied are expected to be of the class Temperature, occupied and expected are expected to be of the class Boolean.»

```
aMessage = #unoccupied
ifTrue:
    [expected
     ifTrue:
        {currentState <= #expectAndNoHeating}
     ifFalse:
        {currentState <= #unoccupiedAndNoHeating}].
aMessage = #current(Temperature | (aMessage = #desiredTemperature)
ifTrue:
    [current <= (desired - 2)
     ifTrue:
        {currentState <= #occupiedAndHeating.
         self startHeating}]
```

Если произошел переход к состоянию `#occupiedAndHeating`, то формируется сообщение, запрашивающее подачу тепла. Соответствующий этому метод `StartHeating` может быть следующим:

```
startHeating
```

```
«Request heat for this room.»
```

```
theHeatFlowRegulator needsHeat: self
```

Метод `startHeating` будет активизироваться во всех случаях, когда произойдет переход из необогреваемого состояния в обогреваемое. В обратном случае активизируется метод `stopHeating` (прекращение обогрева).

Реализация класса-обогревателя (furnace)

Класс `Furnace` реализуется по схеме, аналогичной классу `Room`, в части использования механизма управления состоянием через логику зависимостей языка `Smalltalk`. Структура класса-обогревателя имеет следующий вид:

```
Name:           Furnace
Cardinality:    1
```

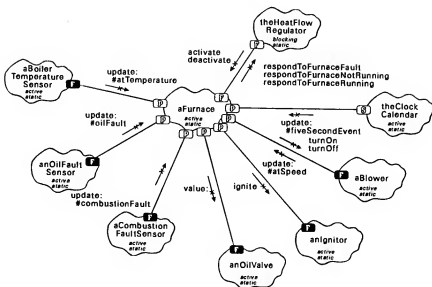


Рис. 8-19. Диаграмма объектов обогревателя.

Hierarchy:	Superclasses:	Object
Public Interface:	Uses:	HeatFlowRegulator
	Operations:	activate deactivate initialize: release
Implementation:	Uses:	BoilerTemperatureSensor ClockCalendar CombustionFaultSensor Blower OilValve Ignitor OilFaultSensor
	Fields:	currentState itsSemaphore theBoilerTemperatureSensor theCombustionFaultSensor theHeatFlowRegulator theBlower theOilValve theIgnitor theOilFaultSensor theView timeDelay
	Operations:	postFault postNotRunning postRunning update: active
Concurrency:		

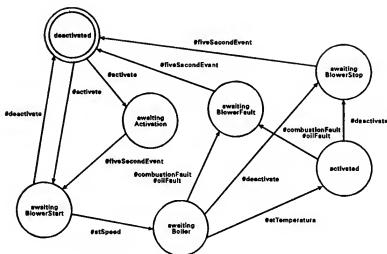


Рис. 8-20. Диаграмма состояний обогревателя.

Приведенная структура соответствует операциям, приведенным на диаграмме объектов (рис. 8-15). Число методов здесь невелико, поскольку механизм контроля обогревателя достаточно прост.

Диаграмма объектов обогревателя показана на рис. 8-19. Так же как и в классе Room, здесь имеется несколько отдельных полей. Каждый из объектов имеет доступ к объекту-печке через механизм зависимостей Smalltalk.

Поток управления в данном объекте аналогичен потоку в объекте-помещении. Все датчики, вентилятор и часы/календарь могут посылать сообщения update: с аргументами, обозначающими природу возникающих событий. Прохождение сообщений может блокироваться в случае их параллельности.

Процесс включения и выключения обогревателя подробно изложен в исходных требованиях. Эти процессы составляют основу поведения обогревателя. На рис. 8-20 показана схема состояний и переходов, соответствующая указанным требованиям. Из исходного (выключенного) состояния с помощью метода activate осуществляется событие #activate, которое переводит печку в состояние awaitActivation или awaitingBlowerStart. Конкретный выбор зависит от того, сколько времени прошло после последней остановки обогревателя. Если обогреватель находился в неактивном состоянии более 5 мин, выбирается состояние awaitingBlowerStart. В противном случае выполняется переход в состояние awaitActivation, в котором каждые 5 с проверяется длительность интервала времени после выключения обогревателя.

Переход в состояние awaitingBlowerStart вызывает включение вентилятора, после чего возникает состояние ожидания сигнала от вентилятора (еще один активный канал управления) о достижении нужной скорости вращения. Здесь мы обнаруживаем неясность исходных требований: состояние ожидания сигнала от вентилятора может длиться неопределенно долго, если требуемая скорость вращения не будет достигнута.

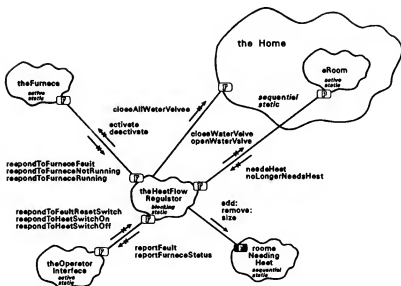


Рис. 8-21. Диаграмма объектов регулятора подачи тепла.

Как только принимается сигнал от вентилятора, открывается клапан топлива и осуществляется воспламенение. После этого вновь возникает состояние ожидания одного из следующих событий: 1) обнаруживается неисправность (переход обогревателя в состояние `awaitingBlowerFault`); 2) обогреватель выключается (переход в состояние `awaitingBlowerStop`); 3) температура воды достигает требуемого значения (обогреватель переходит в состояние `activated`).

Обычный порядок выключения обогревателя состоит в переходах от состояния `activated` к `awaitingBlowerFault` и к `deactivated`. После выключения регистрируется время останова. Значение этого времени сохраняется в поле `timeDelay` и служит при повторном включении для проверки условия минимальной паузы в 5 мин.

Реализация регулятора подачи тепла

Нам осталось реализовать только абстракцию регулятора подачи тепла. Регулятор является ядром системы, поскольку именно он управляет работой обогревателя (включением и выключением) и регулирует подачу нагретой воды в помещения. Тщательная проработка структуры системы и разделение ее на несколько независимых объектов позволяет достаточно удобно реализовать класс регулятора.

На основе уже сделанных проектных решений получим следующую структуру класса `heatFlowRegulator`:

Name: HeatFlowRegulator
Cardinarchy: 1

Hierarchy:	Superclasses:	Object
Public Interface:	Uses:	Furnace Nome OperatorInterface Room
	Operations:	initialize: furnace: operator: needsHeat: noLongerNeedsHeat: release respondToFaultResetSwitch respondToFurnaceFault respondToFurnaceNotRunning respondToFurnaceRunning respondToHeatSwitchOff respondToHeatSwitchOn

Для реализации механизма блокировки в этом классе необходим экземпляр переменной в виде семафора. Регулятор имеет также механизм управления переходом состояний, и, следовательно, должно быть поле для хранения текущего состояния. Для передачи сообщений от регулятора другим объектам необходимо иметь также поля-ссылки на дом, обогреватель и интерфейс оператора.

Требования к системе устанавливают необходимость хранения данных о потребностях помещений в тепле даже при пассивном состоянии обогревателя. Это оправдано, поскольку при активизации обогревателя нужно подать тепло во все помещения, которые направляли соответствующие сообщения. Регулятор должен обеспечивать теплом все помещения, которые этого требуют.

Перечень помещений, нуждающихся в тепле, динамически изменяется; он пополняется при получении сообщений `needHeat` и сокращается после открытия соответствующих клапанов подачи воды (с началом процесса нагревания в помещениях). Если потребуется по какой-либо причине выключить обогреватель (например, во всех помещениях достигнута требуемая температура), информация о перечне обогреваемых помещений сохраняется. Для этого введено поле, учитывающее открытые клапаны.

Диаграмма объектов регулятора показана на рис. 8-21. Диаграмма аналогична диаграмме объектов верхнего уровня (рис. 8-15), но имеет больше подробностей, так как регулятор является ядром управления системой. Однако на рис. 8-21 указан еще один объект `roomsNeedingHeat`, который появляется благодаря принятому подходу к реализации данного класса.

Теперь мы имеем все необходимое, чтобы определить отношения «видимости» между объектами, приведенными на рис. 8-21. В качестве полей регулятора видимыми являются дом, обогреватель и интерфейс оператора. Соответственно видимым является и регулятор для этих объектов.

Сообщения, которые регулятор может посылать объекту обозначенному `roomNeedingHeat`, являются сигналами на включение и исключение помещений из списка, а также определение числа помещений в списке (требующих обогрева) через селектор `size`. Для реализации класса `roomNeedingHeat` можно использовать предопределенный класс `Set`. Для перечня помещений достаточно последовательной семантики. Поскольку сам регулятор имеет механизм блокировки, то для объекта `roomNeedingHeat` не возникает может иметься несколько каналов управления.

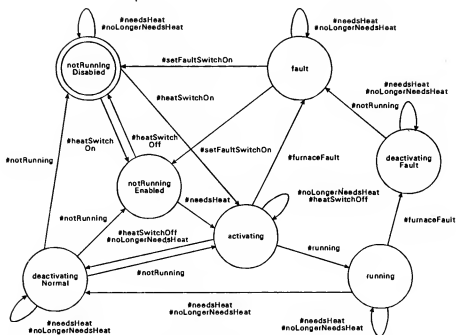


Рис. 8-22. Диаграмма переходов состояний для регулятора подачи тепла.

Управление переходом состояний в регуляторе тепла сосредоточено в методе `update`. В этом нет ничего загадочного: поскольку где-то механизм переходов должен быть реализован, вполне естественно поместить соответствующий код в том же методе, в котором его реализуют другие классы (обогреватель и помещение). Методы реализации переходов состояния под влиянием внешних событий основаны на использовании алгоритмической декомпозиции. Для примера приведем реализацию метода `activating` в следующей форме:

activating: aMessage room: aRoom

Respond to a message while in the activating state. Possible events are #needsHeat, #noLongerNeedsHeat, #HeatSwitchOff, #furnaceFault, and #running.

```
gMessage = #needsHeat
```

```
if True:
```

```
[(roomsNeedingHeat includes: aRoom)
```

```
if False:
```

```
[roomsNeedingHeat add: aRoom].
```

^anil.

```
aMessage = #noLongerNeedsHeat
```

```
if True:
```

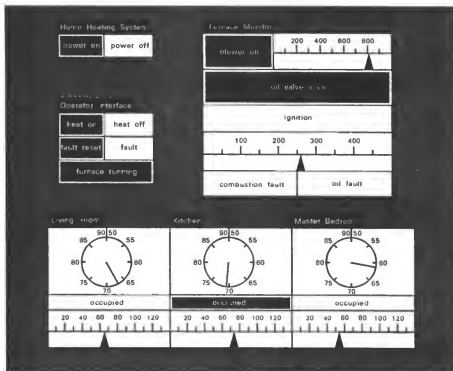


Рис. 8-23. Моделирование системы отопления.

```

[roomsNeedingHeat remove: aRoom ifAbsent: [ ]].
roomsNeedingHeat size = 0
    ifTrue:
        [currentState <= #deactivatingNormal.
         [theFurnace deactivate] fork].
        ^nil.
aMessage = #heatSwitchOff
    ifTrue:
        [currentState <= #deactivatingNormal.
         [theFurnace deactivate] fork.
         ^nil].
aMessage = #furnaceFault
    ifTrue:
        [currentState <= #deactivatingFault.
         [theFurnace deactivate] fork.
         ^nil].
aMessage = #running
    ifTrue:
        [currentState <= #running.

```

```

roomsNeedingHeat do: [:x 1
    x openWaterValve.
    totalValveOpen <— (totalValveOpen + 1)].
roomsNeedingHeat <— Set new.
[theOperatorInterface reportFurnaceStatus: true] fork.
^nil].

```

Полная диаграмма переходов для класса `HeatFlowRegulator` приведена на рис. 8-22. Обратим внимание на то, как на этой диаграмме отражен метод `activating`. В состоянии `activating` (см. диаграмму и код программы) регулятор может получать сообщения о событиях `#needsHeat` и `#noLongerNeedsHeat`. В этих случаях происходит добавление или исключение помещений из перечня. Если перечень становится пустым, то выполняется переход в состояние ожидания сигнала о запуске обогревателя (событие `#running`) с последующим переходом в состояние `running`.

В состоянии `activating` может поступить также сообщение о событии `#heatSwitchOff` (выключение температуры), которое требует отключить обогреватель и перейти в состояние `deactivatingNormal`. В случае неисправности обогревателя (событие `#furnaceFault`) также выполняется его выключение с переходом в состояние `deactivatingFault`. Методы класса-регулятора обладают одной характерной чертой. Метод `activating` room: может посылать сообщение `deactivate` объекту-обогревателю. Это сообщение пересылается не прямо, а через объект-регулятор, что является хорошим примером использования посредника, т.е. объекта, выполняющего операции по сигналу другого объекта. Это сделано потому, что нельзя допустить блокировку регулятора во время отключения обогревателя. В это время могут обрабатываться сигналы о событиях `#needHeat` или `#noLongerNeedHeat` и блокировка регулятора приведет к блокировке указанных сообщений от помещений.

Если попытаться реализовать процесс без посредника, может возникнуть тупиковая ситуация. Регулятор направляет сообщение `deactivate` объекту-обогревателю и блокируется до получения ответа о выключении. Получив от обогревателя сообщение `RespondToFurnaceNotRunning`, регулятор разблокируется. Такая цепь зависимостей может создать тупиковые петли: обогреватель ожидает сигнал регулятора, а регулятор ожидает сигнал от обогревателя. Включение в эту цепь объектов-посредников разрывает такие виды тупиковых петель. Почему нельзя изменить метод `deactivate` так, чтобы при блокировке регулятора обогреватель не мог бы послать сообщение `FurnaceNotRunning`? Если заблокирован регулятор, то блокируются и запросы от помещений на подачу и прекращение подачи тепла. В развешенном процессе разрываются тупиковые петли и повышается степень параллелизма.

Почему мы обратились к вопросу о замкнутых петлях? Из рис. 8-21 очевидно наличие циклических зависимостей между параллельными объектами. Пути передачи сообщений между регулятором и печкой являются двусторонними. В подобном случае проектировщик должен задать вопрос: могут ли сообщения направляться по обоим направлениям одновременно? Если ответ утвердительный, существует потенциальная опасность «гонок» или, что еще хуже, замкнутых петель (тупиков).

8.4. МОДИФИКАЦИЯ

От моделирования к реализации

Мы заканчиваем проектирование домашней системы отопления. В процессе проектирования мы получим пакет диаграмм классов, диаграмм объектов, описаний структур, диаграмм переходов состояний и временных диаграмм. На рис. 8-23 показан снимок с экрана при моделировании на языке Smalltalk нашей системы в полном объеме (включающем 29 специальных классов и примерно 100 методов). Снимок очень похож на прототип интерфейса пользователя. Мы создали систему методом приближения на основе прототипа, а не путем последовательного выполнения этапов анализа, проектирования и реализации. Теперь проект отвечает заданным условиям и остается создать систему с реальными элементами (датчики, обогреватели, переключатели и т.д.). При этом интерфейсную часть классов изменять не потребуется, нужно лишь модифицировать реализацию ряда методов, относящихся к внешнему окаймлению системы.

Посмотрим, как нужно модифицировать класс датчика `Desired-TemperatureSensor`, чтобы он отвечал требованиям реального устройства. Предположим, что реальное оборудование реализует обработку прерываний, и изменим метод `value`: так, чтобы он вызывался по прерыванию в момент изменения пользователем задаваемой температуры. Если система прерываний не реализуется, необходимо организовать постоянный опрос датчика и затем передавать значение `value`. В обоих случаях интерфейс датчика сохраняется. Следовательно, объекты, связанные с этим датчиком, не подвержены влиянию вносимых модификаций.

То же самое относится и к исполнительным элементам, таким, как клапаны подачи воды. Модификация класса `WaterValve` под реальный клапан сводится к изменению реализации метода `value`, чтобы сформировать электрический сигнал управления клапаном. В этом случае интерфейс и логика поведения класса остаются прежними, не затрагивая других проектных решений.

Корректировка исходных требований

В процессе работы мы создали переносной проект (код обычно не переносится). Что произойдет, если мы теперь внесем существенные изменения в исходные требования? Признаком хорошего проекта является его гибкость в отношении подобных изменений.

Предположим, например, что какому-либо заказчику захотелось установить в данной системе отопления иное значение температуры для незаинтересованных людей помещений, причем для каждого помещения свое значение. В исходных требованиях это значение является для всех комнат. Что нужно изменить в нашем проекте?

Придется изменить только реализацию класса `Room` (помещение). Мы введем новый класс `SetBack`, моделирующий устройство задания требуемой температуры в отсутствие людей. В результате это значение будет принадлежать структуре каждого помещения. Механизм управления переходами состояний должен учесть возможность изменения этого параметра. И наконец, алгоритм формирования запросов на подачу тепла в помещение должен будет вместо имевшейся ранее константы использовать значение этого вновь введенного параметра. При этом интерфейс класса `Room` не изменяется: сигналы регулятору формируются по прежнему протоколу. В результате преж-

ние абстракции и механизмы нашего проекта сохранились, лишь добавлено новое свойство в одном фрагменте системы.

Теперь рассмотрим возможность более серьезного изменения требований. Одному из заказчиков в системе отопления необходимо иметь два обогревателя для обслуживания большого здания. Какие изменения следует внести в наш проект?

Изменения будут незначительными. Естественно, потребуется ввести еще один объект класса-обогревателя. Как управлять этим объектом? Регулятор уже имеет все необходимое для такого управления. Например, при превышении числа помещений, имеющих потребность в обогреве, некоторого порога (соответствующего возможностям одного обогревателя) регулятор включает второй обогреватель. Выключение также происходит в связи с уменьшением числа обогреваемых в данный момент помещений. И в этом случае интерфейсная часть всех классов сохраняется, что характеризует наш проект как чрезвычайно стабильный.

Дополнительная литература

Вопросы синхронизации, тупиков, зависаний и гонок подробно рассматриваются Hansen [H, 1977], Ben-Ari [H, 1982] и Holt [H, 1978]. Mellichamp [H, 1983], Glass [H, 1983] и Foster [H, 1981] рассмотрели основные вопросы по системам реального времени. Lorin [H, 1972] изложил проблемы параллельности, связанные с взаимодействием оборудования — программа.

Роль параллелизма в объектном подходе рассматривается в работах, приведенных в гл. 2. Обзор языка Smalltalk с примерами можно найти в приложении.

Глава 9

Object Pascal. Инструментальное средство разработки конструкций геометрической оптики

В то время как Smalltalk является «чистым» языком, в котором все представляется в виде объектов, Object Pascal меньше всего можно считать объектно-ориентированным языком программирования разве что в том смысле, что он добавляет к Pascal только наиболее общую поддержку классов, единственное наследование, динамическую связь и полиморфизм. В отличие от Smalltalk Object Pascal строго типирован.

В данной главе мы используем Object Pascal для решения научных проблем в геометрической оптике, разделе физики, связанном с явлениями отражения и преломления света. При исследовании проблемы были затронуты вопросы синхронизации процесса и разумного распределения поведения между несколькими автономными, относительно статичными объектами. В рассматриваемой ниже проблеме преобладает два очень разных аспекта: с одной стороны, желание иметь дружественный и восприимчивый интерфейс пользователя и, с другой стороны, необходимость интенсивной обработки, использующей динамические объекты.

9.1. АНАЛИЗ

Ограничения

Ниже мы сформулировали подробные требования к инструментальному средству разработки конструкций геометрической оптики. Мы говорили о таких действительных объектах, как линзы и оптические скамьи, а также о таких объектах, как изображения (реальные и мнимые), которые, не будучи явно действительными, на самом деле существуют в нашем представлении как абстракции с точно определенными границами. Что можно сказать в связи с этим о лучах света? Если исходить из волновой теории света, мы могли бы привести доводы о том, что свет не является объектом. Однако с точки зрения геометрической оптики трактование света как объекта (как векторов частиц) является определенным взглядом на мир, потому что он эффективен при моделировании явления рефракции.

Является ли точка фокуса объектом? Точка фокуса определяется как некоторая точка в пространстве на фиксированном расстоянии от центра тонкой линзы. В ходе анализа обнаруживается, что точка фокуса не является хорошим кандидатом в объекты: нет имеющих большое значение операций, связанных с точками фокуса. Лучше моделировать фокусное расстояние как свойство линзы, а точку фокуса как следствие этого свойства. Это аналогично проблеме моделирования физических объектов, таких, как бейсбольные мячи и машины, цвета и формы которых не являются независимыми объектами, но существуют как свойства объектов.

Требования, предъявляемые к инструментальному средству разработки конструкций геометрической оптики

Основная функция данного инструментального программного средства — вычисление пути луча для различных типов тонких линз, расположенных вдоль оптической скамьи. Пользователь может манипулировать такими параметрами, как фокусное расстояние и расположение линз, и сразу же получать в виде обратной связи пути всех главных лучей, размер, позицию и ориентацию результирующих изображений.

Тонкая линза — это линза, толщиной которой можно пренебречь [1]. Нас интересуют два вида тонких линз: собирающие линзы, которые в центре толще, чем по краям, и рассеивающие линзы, которые по краям толще, чем в центре. В собирающей линзе с реальным объектом, помещенным в бесконечности с одной стороны, параллельные лучи, идущие от объекта, приблизятся к линзе, а затем сфокусируются в точке на другой стороне линзы. Эта точка является второй точкой фокуса линзы, которую мы обозначим f' .

Если реальный объект расположен на одной стороне собирающей линзы в первой точке фокуса, которую мы пометим f' , параллельные лучи света появятся с другой стороны линзы. В тонкой линзе величины f и f' равны, и мы называем эту величину фокусным расстоянием линзы.

Если реальный объект не находится ни в бесконечности, ни в точке фокуса, линза сформирует изображение. Пусть d — расстояние от реального объекта до центра линзы, а d' — расстояние от центра линзы до изображения, тогда мы сможем записать следующее равенство:

$$1/d + 1/d' = 1/f.$$

Увеличение m тонкой линзы определяется по формуле:

$$m = -d'/d.$$

Фокусное расстояние собирающей линзы положительно, фокусное расстояние рассеивающей линзы отрицательно. Итак, если дана рассеивающая линза с помещенным в бесконечности с одной стороны линзы реальным объектом, параллельные лучи света от объекта приблизятся к линзе и рассеются. Если мы проследим путь этих рассеянных лучей в обратную сторону, то увидим, что они сфокусируются во второй точке фокуса, которая расположена с той же стороны линзы, что и реальный объект.

Инструментальное средство разработки конструкций геометрической оптики должно быть пригодным для моделирования как собирающих, так и рассеивающих линз. Известны три основных типа собирающих линз:

- * собирающий мениск,
- * плоско-выпуклые линзы,
- * двояковыпуклые линзы.

Существуют также три основных типа рассеивающих линз:

- * рассеивающий мениск,
- * плоско-вогнутые линзы,
- * двояковогнутые линзы.

Если мы имеем реальный объект и одиночную линзу, то данное инструментальное программное средство должно определить размер, место расположения и ориентацию формируемого изображения, которое может быть реальным или мнимым. Например, если реальный объект, располо-

жеи точно позади фокусного расстояния собирающей линзы, реальное изображение объекта будет сформировано с другой стороны линзы. В аналогичном случае рассеивающая линза не сформирует реальное изображение, так как лучи расходятся. Тем не менее изображение объекта появится с той же стороны линзы, с которой расположен реальный объект; мы называем его мнимым изображением. Когда вычисляется путь луча, инструментальное программное средство должно различать реальное и мнимое изображения. Понятие о формировании изображения объекта можно использовать и для набора линз. Мы предполагаем, что все линзы расположены так, что их центры находятся вдоль линии, называемой оптической осью. Таким образом, изображение, сформированное одной линзой, служит объектом для следующей линзы в ряду линз вдоль оптической оси и так далее.

Точка, в которой формируется реальное или мнимое изображение, определяется пересечением только трех лучей, которые называются основными лучами и описываются следующим образом:

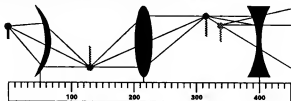


Рис. 9-1. Оптический эксперимент.

1. Луч, параллельный оси, после преломления в линзе, проходит через вторую точку фокуса собирающей линзы или является продолжением луча, выходящего из второй точки фокуса рассеивающей линзы.
2. Луч, проходящий через центр тонкой линзы, не отклоняется, так как две поверхности линзы, через которые проходит центральный луч, практически параллельны друг другу.
3. Луч, проходящий первую точку фокуса (или идущий по направлению к ней), выходит параллельным оптической оси [2].

Рис. 9-1 иллюстрирует эти понятия. Здесь мы видим оптическую скамью с линейной шкалой, значения которой даны в произвольных единицах. На самом левом краю находится реальный объект, за которым расположены три линзы на расстоянии 50, 215 и 400 единиц от реального объекта соответственно. Первая и вторая линзы являются собирающими линзами (их фокусные расстояния имеют значения 30 и 50), а третья линза является рассеивающей линзой (с фокусным расстоянием — 150). Реальное изображение показано темно-серым цветом, а мнимое изображение — светло-серым цветом.

Реальные линзы теряют качество от хроматических и монохроматических аберраций, таких, как сферическая аберрация, при которой основные лучи не сходятся в точку. В данном инструментальном программном средстве нет необходимости моделировать такое явление. Интерфейс пользователя инструментального средства разработки конструкций геометрической оптики должен следовать стандарту интерфейса пользователя реализованного на PC Macintosh фирмы Apple [3]. Каждый оптический экспе-

римент должен быть изображен в его собственном окне, которое можно изменять в размерах и передвигать по экрану. Так как конкретный эксперимент может включать много линз, то каждое окно должно быть предназначено для отображения произвольной, определяемой пользователем части оптического эксперимента. Пользователь должен иметь возможность переключать изображение оптической скамьи, линий шкалы (для регулирования точности расположения линз) и страницы прерываний, перемещения мыши на сетке из пяти интервалов, выбрать любую из шести различных тонких линз из набора. Выбранную линзу можно быть поместить с помощью мыши вдоль оптической скамьи. Необходимо обеспечить интерфейс, который позволял бы пользователю устанавливать фокусное расстояние выбранной линзы. Пользователь должен иметь возможность выбрать, переместить, вырезать, скопировать, очистить и вставить как отдельную линзу, так и набор линз. После любых действий пользователя, которые приводят к изменению размера, расположения и ориентации изображения, инструментальное программное средство должно показать путь луча. Реальный объект (единственный и изображенный в позиции 0 вдоль оптической скамьи) должен быть показан черным цветом. Реальные изображения должны быть показаны темно-серым цветом, а мнимые изображения — светло-серым цветом.

Пользователь должен иметь возможность отображать на дисплее до четырех экспериментов за один раз. При этом применяются обычные операции над файлами: пользователь создает новые эксперименты, открывает старые эксперименты, сохраняет эксперименты, сохраняет копии экспериментов, возвращается к сохраненным экспериментам. Пользователь должен также иметь возможность распечатать эксперимент. Функции отмены и переделки должны быть реализованы во всех операциях пользователя, которые изменяют расположение, размер и ориентацию изображения.

Сам по себе оптический эксперимент также является объектом, потому что этот термин обозначает нечто с точно определенными границами. Оптический эксперимент включает набор линз, оптическую скамью, набор изображений (как реальных, так и мнимых) и набор лучей.

Некоторые требования, изложенные выше, повторяют математические постулаты геометрической оптики; остальные требования относятся к семантике интерфейса пользователя, которые описывают, каким образом можно использовать инструментальное программное средство для управления оптическим экспериментом. Несколько требований описывают видимые для пользователя процессы, такие, как удаление, чистка, копирование, вставка, перемещение и выбор. Вместо того чтобы представлять их в виде алгоритмических абстракций, мы можем ввести объекты, которые выступают в качестве посредников, ответственных за осуществление процессов. Итак, вместо того чтобы иметь процесс удаления, мы будем иметь объект-команду «вырезать», которая содержит информацию, как удалить линзы из оптической скамьи. Инкапсулирование такого поведения в объекте обеспечивает сложное взаимодействие с пользователем. Например, с помощью объекта-команды «удалить» — легче всего узнать, как отменить операцию удаления и спасти удаленную линзу, чтобы ее можно было вставить обратно в это приложение или в другие.

Наша задача состоит в том, что мы должны построить инструментальное программное средство, с помощью которого пользователь сможет беспрепятственно манипулировать оптическими экспериментами. Нам не хотелось бы построить враждебный пользователю интерфейс, который заставлял бы его ставить оптические эксперименты и модифицировать их по этапам в порядке, жестко определенном инструментальным программным средством, а не самим пользователем (что характерно для модальных, пакетно-ориентированных приложений). Вместо этого наши требования подводят нас к созданию редактора типа «что вы видите, то и получаете» (what you see is what you get (WYSIWYG)) для оптических экспериментов, который делает явно видимыми ключевые абстракции, существующие в сознании пользователя, — линзы, изображения, лучи и оптические скамьи. Таким образом, после небольшой практики использования инструментального средства разработки конструкций геометрической оптики пользователь сможет манипулировать абстракциями линз так же, как если бы они были реальными объектами. Основное преимущество такого инструментального программного средства заключается в том, что во многих отношениях оно является более гибким, чем манипулирование реальными линзами. Например, наши требования позволяют пользователю легко изменить фокусное расстояние, что в реальной жизни заставило бы искать новую линзу с нужными свойствами или ее изготовить.

Очень важно, что мы отводим время на проектирование восприимчивого и дружелюбного интерфейса пользователя.

Большая библиотека классов

Проектирование интерфейса пользователя — это независимая от приложений технология, для которой были разработаны несколько весьма эффективных библиотек компонентов программного обеспечения многократного использования. Существуют такие коммерческие продукты, как MIT's X Window System [4], Open Look [5], Microsoft's Windows [6], IBM's Presentation Manager [7]. Все системы управления окнами отличаются друг от друга: одни ориентированы на сети, другие — на ядро операционной системы; в одних отдельные точки раstra считаются наиболее простым графическим элементом, в других же используются абстракции высокого уровня, такие, как прямоугольники, овалы, дуги. В любом случае все эти продукты предназначены для одной цели — упростить задачу реализации той части приложения, которая образует интерфейс человек-машина. Следует отметить, что ни один из этих продуктов не появился в один день. Наиболее эффективные системы управления окнами были созданы лишь через определенное время из небольших, испытанных систем. Потребовались годы успехов и неудач в поиске ряда важнейших абстракций, прежде чем возникла индустрия построения интерфейсов пользователя. Мы имеем несколько различных моделей систем управления окнами, так как нет единственно правильного варианта решения проблемы интерфейса пользователя.

Как и Smalltalk, Object Pascal имеет обширную библиотеку классов, представленную в форме Apple's MacApp [8]. Apple характеризует MacApp как «интегрированную систему построения объектно-ориентированных приложений» [9]. Итак, MacApp — это нечто большее, чем система управления окнами, она обеспечивает классы для построения окон, объектов отображения, диалогов и управления, а также содержит классы, представляющие команды, документы и целые приложения. Проще говоря, MacApp состоит из

определенного числа классов и общедоступных процедур, которые реализуют большую часть общего поведения, необходимого для построения приложения, соответствующего Стандарту интерфейса пользователя Macintosh [10]. В действительности MacApp делает сложным — хотя вполне возможным — построение приложений, которые нарушают этот Стандарт.

MacApp включает большое количество исходных текстов. Ее реализация в Object Pascal превышает 40.000 строк исходных текстов, распределенных в объектно-ориентированных библиотеках (которые содержат различные классы и глобальные определения) и в нескольких библиотеках, не являющихся объектно-ориентированными (которые содержат различные общедоступные процедуры, поддерживающие такие общедоступные средства, как управление меню и распределение памяти). Кроме этих библиотек, существует более шестидесяти программных модулей, предназначенных именно для Object Pascal, которые обеспечивают доступ к программам пакета разработчика Macintosh и другим общедоступным средствам. Диаграмма классов для MacApp показана на рис. 9-2 (представлены только отношения наследования между всеми этими классами). Обратите внимание, что базовый класс в MacApp называется TObject, и из этого класса определяются подклассы возрастающей специализации. По взаимной договоренности все классы в MacApp названы T<something>, чтобы отличать их от простых Pascal-типов.

Тот, кто впервые работает с MacApp, часто испытывает трудности, связанные с большим числом классов. Что представляют эти классы? Как они работают вместе? Как их можно использовать в проблемно-зависимой области? Какие классы действительно важны, а какие можно опустить? На эти вопросы мы должны ответить, прежде чем использовать MacApp для построения какого-либо нетривиального приложения. К счастью, необязательно знать все нюансы такой большой библиотеки, как MacApp, так как при программировании на языке высокого уровня не надо понимать, как работает микропроцессор.

Действительно, трудный момент в обращении с любой большой интегрированной библиотекой классов — это изучение того, какие механизмы она содержит. Чем больше вы знаете о ее механизмах, тем проще найти новые методы использования уже существующих компонентов вместо того, чтобы что-то изобретать на голом месте. На практике, как мы заметили, разработчики обычно начинают с использования наиболее явных классов библиотеки. Когда они выходят на определенный уровень абстракций, они все больше приближаются к использованию более сложных классов. В конце концов разработчики могут найти какую-то свою схему работы со встроенным классом и добавить ее к библиотеке как примитивную абстракцию. Группа разработчиков может заметить, что определенные проблемно-зависимые классы часто появляются в системах, они их тоже включают в библиотеку классов.

Именно так со временем увеличиваются библиотеки классов — не в один день, а постепенно.

Анализ основных механизмов MacApp

Мы не преследуем цель — предложить исчерпывающее руководство по MacApp. Тем не менее мы должны разработать концептуальную интегрированную систему для MacApp таким образом, чтобы можно было использовать ее классы в виде основы для проектирования инструментальной системы разработки конструкций геометрической оптики.

состояние каждого элемента меню (например, может или не может выполняться) и действие, которое должно быть выполнено, когда пользователь выбирает данный элемент. Остальные действия MacApp выполняет (даже для иерархических меню), включая управление, перемещение графического курсора мыши, отмену выбора меню и вызов соответствующего действия для выбора меню. Основное внимание разработчик должен сконцентрировать на том, чтобы дополнить систему проблемно-зависимым поведением путем создания новых классов, которые заменяют существующие методы или добавляют новые методы и поля, вместо того чтобы беспокоиться о неприятных моментах, связанных с использованием вызовов нижнего уровня из пакета разработчика Macintosh.

Большинство классов, показанных на рис. 9-2, являются абстрактными классами. Поэтому разработчик вряд ли будет создавать экземпляр встроенного в MacApp класса, вместо этого он создаст только экземпляры подклассов. Во всех наиболее важных случаях (например, для класса TApplication) MacApp выдает сообщение об ошибке, если некоторые методы не заменены.

Чтобы использовать MacApp для создания нетривиального приложения, мы должны понять каким образом работают следующие механизмы:

- * Изображение подобъектов в объекте отображения.
- * Перемещение графического курсора и отклик на действия мыши.
- * Отклик на команды меню.
- * Отклик на событие.
- * Сохранение и восстановление состояния приложения в документе.

Эти механизмы формируют центральные понятия MacApp. Многие другие механизмы также важны для вывода на печать, удаления, копирования, чистки, вставки, взаимодействия с пользователем в интерактивном режиме, восстановления отказа и распределения памяти, но лучше всего их рассматривать в контексте всего приложения.

Изображение подобъектов в объекте отображения. На рис. 9-3 мы видим весьма типичное для Macintosh окно, которое разбито на шесть объектов отображения: собственно окно (window), палитру (palette), две линейки прокрутки (scrollbar), окно просмотра (scroller) и изображение некоторой модели. Данная иерархия является классическим примером отношения: подобъект отображения (subview) — надобъект отображения (superview). В частности, изображение модели является подобъектом отображения окна просмотра, которое в свою очередь является подобъектом отображения окна. Аналогично палитра, горизонтальная линейка прокрутки и вертикальная линейка прокрутки являются прямыми подобъектами отображения окна.

Структура классов этих объектов формирует другую иерархию. Например, окно является экземпляром подкласса класса TWindow, который сам является подклассом класса TView. В конечном счете TView является суперклассом всех шести объектов.

MacApp не накладывает ограничений на формирование структурной вложенности объектов отображения. Это в свою очередь обусловлено тем, что класс TView включает следующие два поля:

- * fSuperView Объект отображения, в который помещен данный объект отображения
- * fSubviews Список всех объектов отображения, которые содержатся в данном объекте отображения

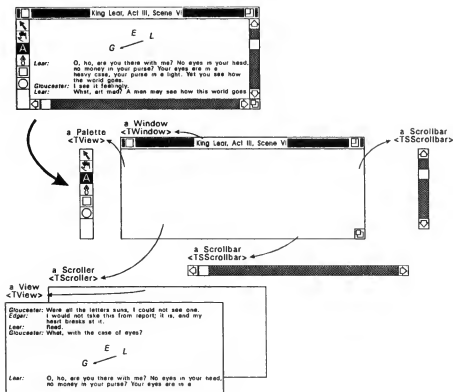


Рис. 9-3. Структурная иерархия окна.

Так как каждый объект отображения, изображаемый на экране, в конечном счете является потомком класса TView, то все такие объекты наследуют эти два поля.

По соглашению в MacApp поле каждого класса называется f<something>. В отличие от Smalltalk язык ObjectPascal является строго типированным, следовательно, каждое поле должно быть объявлено как экземпляр определенного класса или как простой Pascal-тип. Классами упомянутых выше двух полей являются соответственно TView и TList. Так как Object Pascal допускает простой полиморфизм, действительный объект, на который указывает поле, может быть либо непосредственным экземпляром объявленного класса (или типом) или в более общем случае экземпляром подкласса (или подтипом объявленного класса или типа).

Все поля в Object Pascal неинкапсулированы. Например, если мы имеем объект, названный aWindow, любой объект-пользователь, которому доступен

этот объект, может обратиться непосредственно к данному из его полей, например `aWindow.fDocument`. Более того, в отличие от C++ все методы в Object Pascal являются доступными. Использование неинкапсулированных абстракций является менее надежным, потому что оно допускает прямую зависимость объектов-пользователей от реализации нижнего уровня некоторого объекта; если реализация объекта изменяется или мы изменяем значение определенных его полей, мы не можем использовать семантику любого из его объектов-пользователей. По этой причине мы будем избегать прямых ссылок от объекта-пользователя на поля объекта, хотя в некоторых случаях удобство оправдывает риск. Объявление методов класса следует за объявлением его полей. В частности, `TView` имеет широкий, сложный интерфейс: существует 92 метода, определенные для данного класса, не говоря уже о тех методах, которые унаследованы от его суперкласса `TEvtHandler`. Нет необходимости показывать полностью интерфейс `TView`, но будет полезно указать, что каждая из этих операций может быть отнесена к одной из следующих категорий:

- * Методы создания/уничтожения.
- * Методы преобразования координат.
- * Методы управления подобъектами отображения.
- * Методы открытия/закрытия/активизации.
- * Методы управления выбором.
- * Методы задания размера.
- * Методы задания расположения.
- * Методы фокусирования.
- * Методы построения изображения.
- * Методы проверки/аннулирования.
- * Методы управления мышью.
- * Методы управления курсором.
- * Смешанные методы.
- * Методы управления буфером вырезанного изображения.
- * Методы вывода на печать.
- * Методы ресурсов.
- * Методы инспектирования.

Наиболее непростые подклассы класса `TView` добавляют свои собственные поля и методы и заменяют следующие семь методов:

- * `DoHighlightSelection`.
- * `Draw`.
- * `DoSetUpMenus`.
- * `Fields`.
- * `DoMouseCommand`.
- * `DoMenuCommand`.
- * `DoSetCursor`.

Для класса `TStageDirectionsView` как подкласса класса `TView` рассмотрим следующий фрагмент исходного текста:

```
var
    StageDirectionsView: TStageDirectionsView;
begin
    new(StageDirectionsView);
    StageDirectionsView.IView (aDocument, anEnclosingView, gZeroVPT, (100, 100),
                               SizeFillPages, SizeFillPages);
    ...
end;
```

Здесь мы вначале создаем новый объект класса `TStageDirectionsView`, а затем инициализируем его, используя данный документ и надобъект отображения. Третий и четвертый параметры метода `TView` указывают на расположение на экране объекта отображения и его начальные размеры относительно надобъекта отображения. Последние два параметра указывают, что объект отображения имеет переменный размер, округляемый до размера ближайшей страницы.

Объекты отображения не располагаются сами по себе; они в конечном счете инкапсулированы внутри некоторого окна. Могут быть созданы окна произвольной сложности, но в `MacApp` имеются подпрограммы для создания двух наиболее общих типов окон: окон с единственным просматриваемым объектом отображения и окон с просматриваемым объектом отображения и палитрой, подобно тому что изображено на рис. 9-3. Например, при условии что мы уже имеем объект класса `TStageDirectionsView` и другой объект класса `TPaletteView`, мы можем создать окно с палитрой следующим образом:

```
var
    aWindow : TWindow;
begin
    aWindow := NewPaletteWindow (kWindowResource, kWantHScrollBar, kWantVScrollBar,
                                aDocument, StageDirectionsView, PaletteView,
                                kPaletteWidth, kLeftPalette);
end;
```

Первый параметр указывает на идентификатор ресурса окна, связанный с шаблоном ресурсного файла приложения, который описывает начальный размер окна и появление его на экране. Следующие два параметра устанавливают, что мы хотим иметь окно, содержащее объект отображения, который можно просматривать как в горизонтальном, так и в вертикальном направлении. Четвертый, пятый и шестой параметры обозначают документ, просматриваемый объект отображения и палитру соответственно. Седьмой параметр задает ширину палитры и последний параметр указывает, что эта палитра появится с левой стороны окна.

Побочным результатом действия данного метода является создание анонимного объекта класса `TScroller` как подобъекта отображения данного просматриваемого объекта отображения и как подобъекта отображения окна. Это согласуется с рис. 9-3. Так как просматриваемый объект отображения обычно больше, чем объект, который может быть изображен на экране, то этот объект отображения заключается в `TScroller`-объект, который сам является подобъектом отображения объекта окна. Класс `TScroller` включает знания о линейках прокрутки, так что, когда пользователь выбирает стрелку или область линейки прокрутки или перемещает метку линейки прокрутки, объект класса `TScroller` воздействует на просматриваемый объект отображения, переустанавливая его в видимой части окна. Окна и объекты отображения создаются не сами по себе, а некоторым другим объектом. Наиболее часто окна и их объекты отображения создаются объектами документов, которые инкапсулируют состояние модели приложения и поэтому лучше знают, когда создавать окно и его объекты отображений. Таким образом, мы имеем четкое разделение понятий: документы знают, когда создавать окна, но только окна знают, каким образом они будут созданы.

Когда и как объекты, так же как окна и объекты отображения, разрушаются в приложении? В отличие от Smalltalk язык Object Pascal не имеет средств автоматической чистки памяти. Вместо этого программист должен аккуратно вновь использовать память объектов, которые больше не нужны. MacApp выполняет большую часть сложной работы, зная, когда надо убрать определенный объект; например, когда документ закрыт или окно удалено с экрана. Тем не менее разработчик наделяет каждый класс, который инкапсулирует состояние, знанием того, каким образом вновь использовать это состояние. Для осуществления этой задачи в MacApp используется метод `Free`, определенный в основном классе `TObject` и, таким образом, доступный во всех остальных классах. Например, предположим, что мы объявили поле класса `TStageDirectionsView` с именем `fDirections` как экземпляр подкласса класса `TList`. Мы должны выполнить метод `Free` следующим образом:

```
procedure TStageDirections.Free; override;
begin
    fDirections.FreeList;
    inherited Free;
end;
```

Здесь мы сначала освобождаем память, связанную с полем `fDirections`, и затем вызываем соответствующий метод в цепочке объектов суперкласса. Такой стиль программирования вполне обычен для приложений, написанных на объектно-ориентированных языках программирования; тело метода кратко выражено, потому что оно добавляет к суперклассу только описание поведения, специфического для класса. Документы знают лучше, когда создать окно, но только окна знают то, каким образом они создаются. Аналогично этому окна включают знания того, каким образом их изобразить, но не знают, когда это нужно сделать. Вместо этого объект-приложение решает, когда восстанавливать изображение окна, например когда открывается новый документ или когда пользователь установит графический курсор на частично закрытое окно, нажмет и отпустит клавишу мыши для его активизации. В конечном счете объект-приложение осуществляет вызов метода `Update` объекта-окна, который в свою очередь вызывает метод `DrawContents` объекта-окна. Результатом действия метода `DrawContents` является вызов метода `Draw` для самого же объекта-окна, и затем вызов метода `DrawContents` для каждого из его подобъектов отображения. Аналогично изображению строится сверху вниз по связям надобъект отображения/подобъект отображения. В результате рекурсия охватывает все подобъекты отображения, которые являются частью данного окна.

Метод `Draw` должен быть переопределен для каждого проблемно-зависимого объекта отображения; например `TStageDirectionsView` и `TPaletteView`. В классе `TView` данный метод определен, но при этом не выполняет никакого действия, хотя некоторые подклассы имеют текущую реализацию для метода `Draw`, и поэтому они редко переопределяются (например, `TScrollbar`, который изображает на экране стандартную для Macintosh линейку просмотра). Обычно каждый проблемно-зависимый класс объектов отображения содержит поле, указывающее на объекты, которые должны быть изображены на экране в объекте отображения, представляющего его модель. Такое объединение объектов обычно является гетерогенным, т.е. объекты могут принадлежать различным классам. Например, на рис. 9-3 объект отображения включает кроме текста также и линии (и прямоугольники, и окружности согласно па-

литре). В данной ситуации становится полезным полиморфизм: мы можем объявить объект класса TList как объединенные объектов, каждый из которых может принадлежать различным классам, но все они используют общий класс, который является предком некоторого подкласса класса TObject; например, мы можем назвать этот класс TDrawableObject. Исходя из требований, предъявляемых в Object Pascal к семантике типов, мы должны объявить метод Draw в этом общем подклассе, но затем переопределить его в каждом подклассе TDrawable Object для того, чтобы придать специфичное для класса поведение. Таким образом, чтобы изобразить объект отображения, который содержит объединение данных объектов, мы просто применяем итерацию к списку и извлекаем метод Draw для каждого объекта, который мы в нем найдем. Если нам надо изменить наше приложение, так чтобы объект отображения мог изобразить на экране экземпляры новых классов, нам не придется изменять реализацию изображения объекта отображения; вместо этого нам следует только создать новые подклассы класса TDrawableObject.

На диаграмме объектов, изображенной на рис. 9-4, показаны механизмы изображения в MacApp. Ключевыми объектами в данном механизме, являются окно, его подобъекты отображения и модель, изображаемая на экране каждым подобъектом отображения. Для наших целей мы считаем fSubView1 обособленным полем, потому что объект объединения, на который указывает данное поле, не может использоваться ни с каким другим объектом, кроме самого объекта-окна (хотя в соответствии с ограничениями Object Pascal это поле является доступным). Тем не менее содержание данного объединения, т.е. подобъекты отображения окна, являются совместно используемыми объектами, потому что на них могут ссылаться объекты, отличные от родительского объекта отображения. В частности, каждый отдельный подобъект отображения обычно является видимым как поле объемлющего документа. Подобно этому, модель подобъекта отображения является видимой как поле подобъекта отображения, которое обычно является совместно используемым, так как модель может быть видимой более чем в одном объекте отображения.

Диаграмма объектов на рис. 9-4 охватывает статические связи объектов, включенных в механизмы изображения MacApp. Однако вспомним, что диаграмма объектов отражает только какой-то момент постоянно меняющихся событий или конфигурации объектов. Такие объекты, как окна и объекты отображения, создаются и разрушаются в течение цикла существования программы, и, следовательно, диаграммы объектов могут отразить только наиболее интересные формы взаимодействия в данном наборе объектов в какой-то момент времени. По этой причине мы используем временные диаграммы на рис. 9-4, чтобы отразить динамику механизма изображения MacApp.

Для того чтобы изобразить содержание объекта-окна, некоторый объект-пользователь извлекает метод Update объекта-окна, который в свою очередь вызывает метод DrawContents. DrawContents сначала вызывает метод Draw для самого объекта-окна, а затем метод DrawContents для каждого подобъекта отображения, найденного в объединении fSubView1. DrawContents для каждого подобъекта отображения извлекает метод Draw для этого же подобъекта отображения и затем извлекает DrawContents для каждого из его подобъектов отображения и так далее, пока не останется никаких других подобъектов отображения.

В реализации метода Draw каждого объекта отображения мы находим проблемно-зависимое поведение. Например, если объект отображения имеет

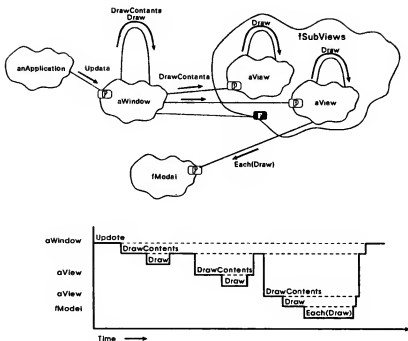


Рис. 9-4. Механизм изображения в MacApp.

поле **fModel**, которое содержит указатель на гетерогенный список изображаемых объектов, то реализация метода **Draw** объекта отображения обычно представляется в виде итерации, когда на каждый объект из данного объединения действует метод **Draw**. Следует определить отдельные объекты, которые появляются в объекте отображения, чтобы включить в них знания о том, каким образом они должны сами себя изображать, вызывая различные **QuickDraw** программы, но не момент, когда они должны быть изображены.

Без изменения основных понятий данного механизма мы можем существенно повысить его производительность. Как уже говорилось выше, модель обычно больше, чем то, что может быть изображено на экране в отдельном окне.

Объект отображения, который содержит такую модель, может быть расширен до нескольких физических страниц, хотя ограничения, связанные с размером экрана, позволяют показать только часть этого объекта отображения в окне. Поэтому мы вводим класс **TScroller**. В частности, объект класса **TScroller** отвечает за отображение части изображения, содержащегося в просматриваемом объекте отображения, как правило, в небольшое окно. Таким образом, когда мы изображаем большой объект отображения, то нет необходимости изображать каждый объект в объекте отображения, так как многие из этих объектов не попадают в видимую часть объекта отображе-

ния. Необходимость в оптимизации возникает тогда, когда процесс изображения объекта становится достаточно сложным и длительным. По этой причине MacApp передает параметр `Area` методу `Draw`. Когда вызывается метод `Draw` для данного объекта отображения, MacApp в первую очередь определяет прямоугольные координаты области, изображение которой должно быть восстановлено. Например, если частично закрытое окно становится активным или если объект-пользователь просматривает объект отображения, MacApp может определить минимальную область объекта отображения, которая была аннулирована и, следовательно, должна быть восстановлена. Значение размеров данной области передается в метод `Draw` объекта отображения, так что при реализации метода `Draw` для каждого объекта отображения можно избежать изображения объекта, находящегося вне аннулированной зоны, путем первой проверки, которая покажет, действительно ли изображение объекта перекрывается значением параметра области. Если это подтверждается, объект изображается; в противном случае ничего не происходит.

Проблемно-зависимый объект-пользователь может использовать тот же механизм. Например, предположим, что мы имеем приложение, в котором пользователь может применять выбранные из палитры сервисные программы для добавления, уничтожения и перемещения объектов внутри объекта отображения. Какой бы объект ни модифицировался в модели объекта отображения, область, содержащая объект воздействия, должна быть известна объекту отображения путем использования одного из методов аннулирования объекта отображения (такого, как метод `InvalidRect`, определенный в классе `TView`). В результате действия данного метода должны скапливаться области объекта отображения, которые устарели и поэтому должны быть восстановлены; параметр области, переданный методу `Draw` объекта отображения, представляет собой объединение всех этих аннулированных областей.

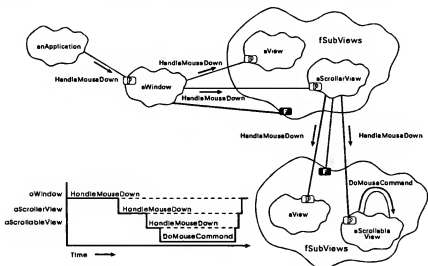


Рис. 9-5. Механизм действия мыши в MacApp.

Перемещение графического курсора и реакция на действия мыши. В таких приложениях, как инструментальное средство разработки конструкций геометрической оптики, предоставляется широкий выбор взаимодействий с пользователем. Выше мы говорили, что класс `TView` является подклассом класса `TEvtHandler`, который содержит знания о том, как реагировать на действия мыши. Возвращаясь к рис. 9-3, предположим, что пользователь нажал и отпустил клавишу мыши где-нибудь внутри просматриваемого объекта отображения, чтобы выбрать абзац для удаления. Механизм, с помощью которого приложение реагирует на это действие мыши, показан на диаграмме объектов на рис. 9-5.

Из диаграммы мы видим, что событие нажатия клавиши мыши в первую очередь обнаруживает объект-приложение, используя механизм основного цикла событий, который мы опишем ниже подробно. Событие нажатия клавиши мыши может произойти в одном из нескольких положений графического курсора на экране:

- * На линейке меню.
- * Вне окна приложения.
- * Внутри контура кнопки, отвечающей за увеличение окна.
- * Внутри контура кнопки, отвечающей за выход из окна.
- * Внутри контура кнопки, отвечающей за увеличение размеров окна до размеров экрана.
- * Внутри содержимого окна приложения.

Механизм реакции `MacApp` на событие нажатия клавиши мыши в положении графического курсора на линейке меню мы опишем ниже. `MacApp` автоматически управляет событиями нажатия клавиши мыши в следующих четырех положениях графического курсора, хотя текущая реакция может быть отменена. Для того чтобы управлять событиями нажатия клавиши мыши в окне приложения, пользователь должен обеспечить проблемно-зависимую обработку этих событий.

Механизм изображения в `MacApp` дает возможность каждому подобъекту отображения, который содержится в окне, изобразить самого себя, начиная от вершины цепочки надобъект отображения/подобъект отображения и двигаясь вниз, пока все объекты отображения не будут использованы. В `MacApp` механизм действия мыши совсем другой: в нем осуществляется поиск самого нижнего объекта отображения в цепочке надобъект отображения/подобъект отображения, который пересекается с расположением графического курсора, и только этому объекту отображения предоставляется возможность отреагировать на действие мыши. Как показано на рис. 9-5, если объект-приложение обнаруживает, что событие нажатия клавиши мыши происходит, когда графический курсор находится внутри содержимого окна приложения, то приложение извлекает метод `HandleMouseDown` для соответствующего объекта-окна. Если это окно не является внешним окном (для простоты это не показано на временной диаграмме), `HandleMouseDown` сначала выбирает окно, чтобы сделать его активным. Для того чтобы зафиксировать подобъект отображения самого низкого уровня, который содержит точку положения графического курсора при нажатии клавиши мыши, `HandleMouseDown` проходит по цепочке надобъект отображения/подобъект отображения, используя итератор `LastSubviewThat`, определенный в классе `TView`. Так как объекты отображения могут иметь подобъекты отображения, как в иерархии окно/окно просмотра/просматриваемый объект отображения, показанной на рис. 9-5, то `HandleMouseDown` вызывается для каждого вложенного объекта отображения.

Если `HandleMouseDown` достигает подобъекта отображения, который пересекается с расположением графического курсора мыши в момент действия мыши, и если подобъект отображения находится на нижнем уровне цепочки надобъект отображения/подобъект отображения, то метод `DoMouseCommand` извлекается только для этого объекта отображения. Как только мы встречаем подобъект отображения, который удовлетворяет этим критериям, поиск по цепочке надобъект отображения/подобъект отображения сразу же приостанавливается. Проблемно-зависимое поведение должно быть заложено в методе `DoMouseCommand` в такой же степени, как и в методе `Draw` для механизма изображения `MacApp`. В классе `TView` реализация метода `DoMouseCommand` по существу отсутствует. Способ замены данного метода зависит от конкретных требований данной проблемной области. Например, если объектом отображения является палитра, обычным действием метода `DoMouseCommand` будет отмена выделения картинки последней выбранной сервисной программы и затем выделение картинки сервисной программы, которая оказывается под графическим курсором. Если объект отображения является просматриваемым объектом отображения, который содержит текст и графику (рис. 9-3), то обычно действие мыши представляет собой выбор или отмену выбора определенных объектов, начало перемещения всех выбранных пользователем объектов или начало применения только что выбранной сервисной программы.

Типичным для `MacApp` является то, что метод `DoMouseCommand` должен создавать объекты команды в виде посредников для выполнения действительной работы. Например, вместо того чтобы действительно перемещать выбранные в данный момент объекты, метод `DoMouseCommand` может создать объект `DraggerCommand` как экземпляр проблемно-зависимого подкласса класса `TCommand`. Создание посредников команды намного выгоднее, чем выполнение работы непосредственно в объекте отображения. Во-первых, это обеспечивает большее разделение функций, позволяя объектам отображения сосредоточиться на изображении и обнаружении событий, а объектам-командам сосредоточиться на выполнении, изменении и отмене действий. Во-вторых, мы достигаем более высокой степени многократного использования и таким образом пишем меньше исходных текстов: поведение, заключенное в объектах-командах, часто необходимо как для механизма команд меню, так и для механизма действия мыши. В-третьих, данный подход является более гибким. Например, если бы мы захотели изменить визуальную обратную связь во время перемещения выбранных объектов, то нам не пришлось бы изменять каждый класс объектов отображения, но, вероятно, мы должны были бы модифицировать определенные методы объекта-команды.

Отклик на команды меню. Когда приложение обнаруживает, что нажатие клавиши мыши произошло в положении графического курсора внутри линейки меню, то `MacApp` реагирует на это с помощью механизма команд меню. Данный механизм не может использовать те же отношения надобъект отображения/подобъект отображения, какие использует механизм действия мыши, в первую очередь потому, что всем объектам, отличным от объектов отображения и окон, должна быть предоставлена возможность реагировать на определенные команды меню. В частности, объекты отображения и окна ничего не знают о том, как открываются и закрываются документы, но документы и приложения знают это; таким образом они должны участвовать в механизме команд меню. Механизм команд меню использует абстракцию, называемую цепочкой команд, изображенной на диаграмме объектов на рис.

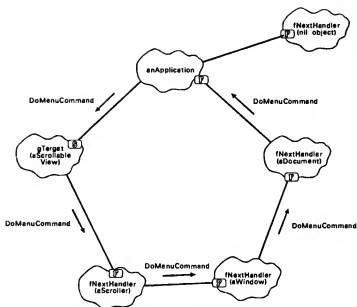


Рис. 9-6. Механизм команд меню в MacApp.

9-6. При формировании цепочки команд используется поле `fNextHandler`, определенное в классе `TEvtHandler`. Таким образом, каждый подкласс `TEvtHandler`, включающий объекты отображения, документы и приложения, имеет данное поле. Всякий раз, когда такой объект инициализируется, в MacApp строится цепочка команд. В частности, каждый подобъект отображения указывает на свой надобъект отображения, каждое окно — на документ, содержащийся в нем, каждый документ — на свой объект-приложение. Таким образом, цепочка команд представляется значительно более сложной, чем список надобъект отображения/подобъект отображения. Эта цепочка включает не только объекты, отличные от объектов отображения и окон, но, кроме того, на каждый объект-приложение могут ссылаться составные документы, на каждый документ — составные окна и на каждое окно — составные объекты отображения.

Целью формирования цепочки команд является объединение объектов, которые могут реагировать на команды меню и упорядочены по их наилучшему положению для выполнения действия. Сначала идут конкретные подобъекты отображения, за ними следуют их надобъекты отображения, затем окно, в котором они содержатся, затем их документ и в конце следует объект-приложение. Отметим, что цепочка начинается не с объекта-приложения, а с объекта значительно более низкого уровня. Головной объект цепочки содержится в глобальной переменной `gTarget` (по соглашению в MacApp все глобальные переменные называются `g<something>`). Всякий раз, когда ак-

тивизируется окно, в `gTarget` обычно помещается такой подобъект, который является наиболее интересным для команды меню. Для окна, которое включает составные подобъекты отображения, объект-пользователь указывает на этот подобъект отображения, когда создается окно. Общедоступная процедура `NewPaletteWindow`, например, указывает на просматриваемый объект отображения как на наиболее интересный для команды меню. В связи с этим некоторым объектам отображения обычно не предоставляется возможность участвовать в цепочке команд. Например, объект отображения «палитра», вероятно, никогда не будет частью цепочки команд, так как он почти не реагирует на какую-либо команду меню.

Когда нажатие клавиши мыши произошло на линейке меню, то метод `HandleMouseDown` объекта-приложения сначала вызывает программу `MenuSelect` из Пакета разработчика Macintosh для того, чтобы определить, какие были выбраны меню и его элемент. Данная общедоступная процедура перемещает графический курсор мыши, выделяет все элементы меню, спускает вниз меню и в конце возвращается к прежнему состоянию, когда отпускается клавиша мыши. Если пользователь отпустил клавишу внутри элемента меню, объект-приложение вызывает свой метод `MenuEvent`, для того чтобы определить, какое меню и в какой элемент были выбраны, и закодировать данную пару меню/элемент пару единственным числом, называемым командным числом. Как показано на рис. 9-6, приложение затем вызывает метод `DoMenuCommand` для объекта, указатель на который содержится в глобальной переменной `gTarget`. Если данный объект не имеет возможности отреагировать на команду, такое же сообщение посылается следующему объекту в цепочке. Этот процесс продолжается до тех пор, пока не будет найден объект, который сможет отреагировать на команду меню, либо `fNextHandler` не окажется нулевым (что в MacApp является текущей величиной для данного поля для всех объектов-приложений).

Пользователь должен поработать с методом `DoMenuCommand` для того, чтобы вложить в него проблемно-зависимое поведение. Обычно реализация данного метода представляется большим количеством операторов выбора:

```
begin
  DoMenuCommand := gNoChanges;
  case ACmdNumber of
    cSelectAll:    ...
    cCut:          ...
    cCopy:         ...
    cPaste:        ...
    cClear:        ...
    kDrawingSize: ...
    otherwise DoMenuCommand := Inherited DoMenuCommand(ACmdNumber);
  end;
end;
```

В MacApp все командные константы именуются `<something>`, а другие типы констант — `k<something>`. Мы решили именовать все проблемно-зависимые константы как `k<something>`.

Команда `DoMenuCommand` должна проверять, какие еще команды могут правильно управляться всеми объектами заданного класса. Например, просматриваемый объект отображения должен иметь возможность отреагировать на команды выбора, вырезания, копирования, вставки, чистки и изменения размера изображения. Документ, с другой стороны, должен только от-

реагировать на команды вывода на печать, сохранения и возвращения. Если указанный объект не может управлять командой, как показано в предшествующем исходном тексте, то надо просто вызвать соответствующий метод в суперклассе. Если суперкласс не может отреагировать на данную команду, то же самое делается с его суперклассом и т.д. Предположим, что мы не нашли промежуточный суперкласс, который может отреагировать на команду меню. Управление в итоге достигает метода, определенного в классе TEviHandler, в котором реализован вызов метода DoMenuCommand для следующего объекта в цепочке команд. Таким образом, прохождение по цепочке заканчивается, когда мы достигаем объекта, чье поле fNextHandler имеет нулевое значение (которое является обычной величиной устанавливаемой в объекте-приложении). Редко прохождение по цепочке достигает конца, потому что это означало бы, что мы имеем команду, для которой не существует реагирующего на нее объекта.

Имея такой механизм, проектировщик может сконцентрироваться на желаемом проблемно-зависимом поведении для каждого класса и быть уверенным в текущем поведении для всех общих команд, таких, как вывод на печать, открытие, закрытие и сохранение документов. Так же как и для механизма действия мыши, мы обычно предполагаем для метода DoMenuCommand возможность создавать объекты-команды как посредники, которые выполняют действительную работу, хотя, с другой стороны, имеется альтернатива выполнять работу на месте. Мы обычно не создаем объекты-команды для простых действий, таких, как выбор; мы используем их для сложных действий, таких, как вырезание, копирование, вставка и чистка. Если мы хотим иметь возможность отменять или выполнять заново действие, лучше всего создавать объект-команду. Если мы хотим использовать действие в других местах или если действие особенно сложное, мы также должны создать объект-команду.

Стандарт интерфейса пользователя Macintosh разрешает иметь эквиваленты элементам меню на клавиатуре. Например, комбинация командных клавиш клавиатуры Command-x обычно обозначает элемент Cut в меню Edit. При реакции на нажатие командной клавиши клавиатуры повторно используется механизм команд меню. В частности, когда приложение обнаруживает комбинацию командных клавиш клавиатуры, вызывается метод DoCommandKey вдоль всей цепочки команд. В текущей реализации метод DoCommandKey ничего не выполняет, но при этом происходит переход управления к следующему объекту так, что управление в итоге достигает объекта-приложения, который включает перекодировку командного числа в пару меню/элемент и затем вызывает метод DoMenuCommand, начиная с gTarget и используя командное число. При реакции на команды клавиатуры, отличающиеся от комбинации командных клавиш клавиатуры, также используется цепочка команд. В этом случае приложение вызывает метод DoKeyCommand, который применяется к объекту, указатель на который содержит в gTarget.

Отклик на событие. Существует восемь внешних событий, на которые может реагировать приложение MacApp:

- * Клавиша мыши отпускается.
- * Клавиша мыши нажимается.
- * Окно активизируется.
- * Окно обновляется.
- * Клавиша клавиатуры нажимается.

- * Диск.
- * Система.
- * Аппет.

Многие из этих событий автоматически управляются MacApp. В частности, MacApp обнаруживает и реагирует на события активизации и события обновления окна, события вставки и удаления диска, MultiFinder-события и сетевые события. Событие, связанное с отпусканием клавиши мыши, обычно игнорируется приложениями, кроме тех случаев, когда пользователь перемещает мышь, два или три раза быстро нажимает клавишу, или отпускает клавишу мыши над определенными объектами управления.

MacApp обнаруживает все эти события как часть механизма главного цикла событий. Этот механизм выступает в качестве общепринятой структуры для большинства приложений с однородным интерфейсом, т.е. таких приложений, в которых на порядок взаимодействия пользователя с системой мы накладываем мало ограничений. В данном механизме главная нить управления для приложения состоит из цикла, в течение которого мы сначала либо получаем, либо ожидаем событие. Если событие обнаружено, мы вызываем соответствующую операцию для управления им. Цикл завершается только тогда, когда пользователь выходит из приложения.

В MacApp экземпляры класса TApplication действуют как посредники, отвечающие за механизм событий главного цикла. Поэтому, вместо того чтобы строить структуру корня нашего приложения, как если бы она находилась на вершине алгоритмической декомпозиции, мы в первую очередь используем главную программу для объявления экземпляров подкласса TApplication, а затем вызываем его метод Run (который в конечном счете выводит действие из главного цикла). Преимущество использования объекта-приложения по сравнению с применением алгоритмической абстракции в корне приложения заключается в том, что это дает нам последовательную концептуальную модель, так как все остальное в нашем приложении состоит из наборов совместно работающих объектов.

Как мы уже говорили в гл. 3, мы при проектировании обычно создаем классы, которые экспортируют примитивы. По этой причине в метод Run не заключается непосредственно исходный код для механизма главного цикла; его работа разделяется по нескольким методам. Это позволяет разработчику легко подгонять текущее поведение объекта-приложения. Короче говоря, результатом действия метода Run является вызов метода MainEventLoop. При каждом проходе через цикл MainEventLoop сначала вызывает метод PollEvent объекта-приложения, который ищет новые события. Если событие найдено, MainEventLoop вызывает метод GetEvent для получения всей информации о событии и затем вызывает метод HandleEvent. HandleEvent в свою очередь вызывает метод DispatchEvent объекта-приложения, который посылает метод (такой, как HandleMouseDown), соответствующий определенному типу события.

Приложения, в которых преобладает взаимодействие человек/машина, часто включают относительно длинные периоды времени в течение которых в приложении ничего не выполняется. Пользователь часто тратит много времени, думая о том, что ему делать дальше. В течение этого времени большие компьютерные циклы являются пустыми. Для того чтобы позволить приложению выполнять полезную работу в течение этих холостых периодов, механизм главного цикла события применяет холостую цепочку. Холостая цепочка является простым списком объектов, каждый из которых является

экземпляром некоторого подкласса класса `TEviHandler`. Глобальная переменная содержит указатель на заголовок данного списка, и объекты связываются через поле `fNextHandler`, как определено в классе `TEviHandler`. То же поле используется для формирования цепочки команд, но так как цепочка команд и холостая цепочка являются ортогональными концепциями, данный объект никогда не появляется в обеих цепочках одновременно.

Если нет ждущих событий, механизм события главного цикла проходит по холостой цепочке, вызывая метод `DoIdle` для каждого объекта, который он встречает, так что этот объект всякий раз, когда он пожелает, может выполнить проблемно-зависимую обработку. Каждый объект в цепочке также имеет поле `fIdleFreq` (определенное в классе `TEviHandler`), которое указывает, как часто метод `DoIdle` должен вызываться. Например, если в приложении требуется объект отображения с несколькими мигающими элементами, то мы, вероятно, захотим, чтобы соответствующий метод `DoIdle` вызывался только через каждые несколько интервалов таймера, а не каждый холостой цикл.

Сохранение и восстановление состояния приложения в документе. Пользователю инструментального средства построения конструкций геометрической оптики должна предоставляться возможность создавать эксперименты, манипулировать ими и сохранять их для дальнейшего использования. Тем не менее `MacApp` не включает ничего-слишком сложного, подобно объектно-ориентированной базе данных, и поэтому непосредственно не поддерживает постоянство объектов. Вместо этого мы должны использовать другой механизм, чтобы обеспечить иллюзию постоянства.

`MacApp` использует класс `TDocument` для сохранения состояния приложения. Теоретически документ служит в виде гетерогенного хранилища некоторых объектов-приложений, которые нужно сохранить между реализациями. Например, документ для инструментального средства разработки конструкций геометрической оптики должен представлять единственный оптический эксперимент, состоящий из всех линз, расположенных вдоль оптической скамьи. Документ также должен сохранять состояние характеристик приложения, выводимых на печать, заметок пользователя или расположения и размер всех активных в настоящий момент окон.

Хотя `TDocument` является подклассом класса `TEviHandler`, он добавляет несколько новых полей, таких, как `fTitle` (свое имя), `fWindowList` (список окон, принадлежащих документу) и `fChangeCount` (число изменений в состоянии документа с момента его последнего сохранения). Обычно вместо использования класса определяется его подкласс, который включает также поле для каждого значительного подобъекта отображения и поле, которое указывает на постоянное состояние документа. Мы связываем это состояние скорее с документом, чем с объектом отображения, который изображает его на экране, потому что объекты отображения имеют временный характер, тогда как документы существуют в течение значительного времени жизни модели. Так как эта же модель часто разделена на более чем один объект отображения, то помещение модели вместе с ее документом обеспечивает устойчивую точку ссылки для каждого временного объекта отображения.

В `MacApp` программа может иметь только один объект-приложение, хотя каждый объект-приложение может использовать несколько, возможно, различных объектов-документов. Объект-документ обычно создается всякий раз при запуске приложения или когда пользователь выбирает `New` или `Open` элементы меню в стандартном `File` меню. В результате каждое из этих действий кончается обращением к методу `DoMakeDocument` приложения. Про-


```
begin
    inherited DoRead(ARefNum, RsrcExists, ForPrinting);
    fDirections.DoRead(ARefNum);
end;
```

Мы вызываем унаследованный метод DoRead, чтобы дать MacApp возможность прочесть его состояние (такое, как выбор принтера). Так как состояние документа обычно включает объединение гетерогенных объектов, то при сохранении документа должен кодироваться класс каждого объекта перед регистрацией его состояния. Если метод DoRead поля fDirections вызывается, его действие может быть представлено следующим образом:

```
<read a value indicating the number of objects that follow>
for Index := 1 to Count do
    begin
        <read a value indicating the class of the object that follows>
        <clone a copy of an object of this class>
        AnObject.DoRead(ARefNum);
    end;
```

Данный подход аналогичен механизму изображения, в котором действительная работа в конечном счете выполняется объектами самого нижнего уровня. Как мы увидим ниже, парадигма имитации объектов из прототипов полезна и может быть использована для создания других объектов, таких, как общедоступные процедуры, определенные в палитре.

Здесь, мы рассмотрели только наиболее общие классы в MacApp. Например, мы еще не изучали класс TView с его подклассами и механизмами, которые вместе обеспечивают характеристики простого текстового редактора, поддерживающего многочисленные шрифты, изменение размеров и различные стили. Мы не изучали механизмы, заключенные в классе TGridView, классы, поддерживающие отладку и восстановление ошибок; не были представлены классы, связанные с диалогами. Различия между диалогом и простым окном становятся расплывчатыми в MacApp. Кроме того, представленные ниже простые механизмы являются основными частями для всех этих классов. Это не удивительно, и как мы уже говорили в гл. 1, системы, имеющие сложное поведение, часто конструируются из нескольких относительно простых механизмов. Это в определенной степени верно для MacApp. Мы должны указать также, что существуют ограничения при использовании MacApp. В частности, действительные методы управления памятью в MacApp ограничивают число объектов, которые могут быть созданы в приложении, лишь несколькими тысячами в зависимости от сложности каждого объекта. Поскольку MacApp пригоден для реализации таких приложений, как инструментальное программное средство разработки конструкций геометрической оптики, другие методы должны быть использованы для таких приложений, как инструментальные средства машинного проектирования или программы моделирования геометрических тел, которые могут содержать десятки тысяч объектов. Это не означает, что мы должны отказаться от объектного подхода или отказаться полностью от использования MacApp; это означает только, что мы иногда должны полагаться на смешанное проектирование. В частности, вместо того чтобы трактовать каждый концептуальный объект как конкретный экземпляр подкласса класса TObject, мы должны использовать простые типы Pascal для формирования модели приложения, хотя мы должны еще иметь возможность использовать классы MacApp для формирования интерфейса пользователя.

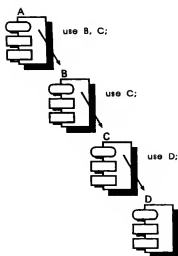


Рис. 9-8. Модульная архитектура ObjectPascal.

9.2. ПРОЕКТИРОВАНИЕ

Модульная архитектура

Декомпозиция приложений Macintosh. В Smalltalk класс — единственно возможный элемент декомпозиции. Хотя программа просмотра системы в Smalltalk позволяет привязать классы к различным категориям, эта концепция применима далеко не во всех случаях, так как она не ограничивает видимость класса внутри группы. Следовательно, все классы, объявленные в среде Smalltalk, глобально видны. Для малых систем это не имеет особого значения, но с переходом к более крупным приложениям мы достигаем уровня, на котором отсутствие модульной структуры делает наш проект очень трудным для понимания. Без средств группировки абстракций в большие структуры мы быстро достигаем пределов человеческих способностей понимать одновременно множество различных предметов.

К счастью, Object Pascal даст структуру для декомпозиции больших систем в отдельные модули, позволяя нам скрыть определенные классы и объекты друг от друга и уменьшить общие зависимости абстракций. Это особенно важно в строго типизируемом языке. Если не уменьшить зависимости абстракций друг от друга, то мы будем вынуждены постоянно recompilировать почти всю систему при внесении малейших изменений. В больших системах, находящихся в процессе разработки, это может существенно повлиять на графику разработки и даже затруднить внесение изменений. Использование модулей для группировки абстракций и решения вопроса о видимости в конечном счете укрепляет нашу уверенность в стабильности и корректности систем.

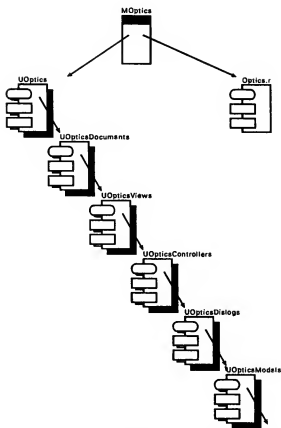


Рис. 9-9. Модульная диаграмма инструментального средства разработки конструкций геометрической оптики.

В Object Pascal синтаксическая структура модуля называется программным модулем. Программный модуль имеет две части — интерфейс и реализацию. Интерфейс программного модуля может содержать только константы, типы, переменные, функциональные и процедурные спецификации, а любые методы или общедоступные процедуры в интерфейсе программного модуля должны быть завершены в его реализации. Итак, интерфейс дает внешнее представление об абстракции, а его реализация содержит внутреннее представление.

В реализацию программного модуля могут быть помещены объявления, но они видимы только в реализации данного конкретного программного модуля и, следовательно, скрыты от других. В частности, мы можем объявить класс в реализации программного модуля так, чтобы он не мог быть отнесен к какому-либо другому программному модулю. И наоборот, любые объ-

явления, помещенные в интерфейс программного модуля, видны в реализации этого программного модуля, а также во всех программных модулях, использующих данный программный модуль. Эти моменты показаны на рис. 9-8. Если реализация программного модуля С требует объявления D, то С должен использовать D. В отличие от Ada в Object Pascal зависимости представлены только в интерфейсе программного модуля, даже если использованный программный модуль необходим только для реализации. Предположим, что метод, объявленный в интерфейсе программного модуля В требует обозначения класса или типа, объявленного в интерфейсе программного модуля С. В должен использовать С, но не должен использовать программный модуль D. Если интерфейс программного модуля А должен иметь доступ к объявлениям из интерфейса программного модуля В, то А тоже должен использовать В. Однако, так как интерфейс В зависит от С, программный модуль А должен также использовать программный модуль С.

В MacApp представлена вторая модульная структура, необходимая для Macintosh. В частности, каждое приложение состоит из двух частей: ветвления данных и ветвления ресурсов. Документы, связанные с приложением, как правило, используют ветвление данных для сохранения своего состояния. Приложения же обычно используют только ветвление ресурсов для сохранения своего кода, а также других ресурсов, таких, как меню, шрифты, пиктограммы, изображения и шаблоны окон. Преимущество этих ресурсов состоит в том, что они позволяют менять внешний вид приложения без перекомпилирования кода исходного текста. Например, предположим, мы первоначально написали программу, предполагая, что ее будут использовать только те, кто владеет английским языком, а теперь мы хотим распространить версию этого приложения во Франции и в Японии. Если мы уже жестко закомпилировали имена пунктов меню или текст в диалоговых окнах, то нам придется изменить код исходного текста и все перекомпилировать. Это не только прибавит проблем в управлении конфигурацией системы и в контроле версий, но также уменьшит нашу уверенность в корректности каждой новой версии, так как по неосторожности мы могли внести изменения в какую-либо одну версию, а в другие нет. Если вместо этого мы поместим эти ресурсы в ресурсный файл, нам придется всего лишь модифицировать ресурсы приложения для изменения его внешнего вида без воздействия на код исходного текста.

Как наличие этих модульных структур отражается на нашем проекте? Мы можем просто не обращать на них внимания и объявить все в одном программном модуле. Это не очень хорошая идея, так как она не учитывает проблемы видимости, с которой мы сталкиваемся в Smalltalk. Мы могли бы использовать другую возможность и поместить каждый класс в отдельный программный модуль. Это тоже не будет оптимальным вариантом: большие системы будут содержать сотни, если не тысячи таких программных модулей, что сделает весьма затруднительным даже нахождение интересующих нас классов. Более того, с помещением классов в отдельные программные модули значительно возрастает сложность системы, потому что приходится распутывать очень сложное переплетение зависимостей программных модулей.

Приемлемый ответ находится где-то посередине, между этими двумя крайностями, а методика, изложенная в гл. 4, может помочь нам прийти к хорошо разработанной модульной декомпозиции. На практике мы убедились, что наилучшим решением является создание модулей, организованных вок-

руг значимых объединений классов и объектов. Под значимыми мы понимаем такие объединения, где каждая абстракция в группе более тесно связана с абстракциями данной группы, нежели с другими абстракциями за пределами группы.

Декомпозиция инструментального средства разработки конструкций геометрической оптики. На рис. 9-9 показан наш проект модульной архитектуры для инструментального средства разработки конструкций геометрической оптики. Наше приложение подразделяется на главную программу (MOptics), ресурсный файл (Optics.r.) и шесть программных модулей. Поскольку мы ожидаем, что каждый из этих программных модулей будет большим, мы можем использовать директиву компилятора, чтобы заставить их код располагаться в отдельных сегментах памяти. Выделение сегментов памяти — это, по всей видимости, отдельный вопрос по отношению к вопросу о выборе структуры классов и объектов, и если включить его в проект модульной архитектуры, это позволит правильно рассуждать об альтернативных подходах к сегментации без воздействия на любую другую часть нашего проекта.

Различное расположение модулей в этой архитектуре вытекает из двух факторов. Первый — это прошлый опыт. Построив несколько дюжины приложений MacApp, мы перебрали много различных модульных архитектур. Некоторые из них работали хорошо, некоторые — не очень, и модульная архитектура, изображенная на рис. 9-9, воспроизводит нашу последнюю каноническую декомпозицию. Это пример повторного использования проекта. Если оглянуться назад, все, что сработало хорошо, — это результат следования принципам тщательной и серьезной разработки. Из этого следует второй фактор: когда мы разлагаем систему на модули, мы применяем основной критерий — разделение на различные области. Абстракции должны быть помещены в отдельные модули, если они связаны не прочно. Более того, если какое-то проектное решение может измениться, лучше всего скрыть его, чтобы оградить другие абстракции от последствий каких-либо изменений.

Итак, на нижнем уровне архитектуры расположен модуль UOpticsModels, который содержит все классы и объекты, связанные с наиболее существенными элементами оптического эксперимента. Этот программный модуль инкапсулирует большую часть проблемно-зависимого поведения приложения; любой другой программный модуль в первую очередь связан с интерфейсом пользователя. Следующим в иерархии модулей является программный модуль UOpticsDialogs, который содержит все средства, необходимые для представления диалогов для взаимодействия с пользователями. UOpticsControllers построен над двумя программными модулями — UOpticsDialogs и UOpticsModels, и в этом программном модуле заключены все классы и объекты, относящиеся к командам пользователя. Далее следуют программные модули UOpticsViews, UOpticsDocuments и UOptics. Мы не объединяем эти программные модули, поскольку они содержат три различных механизма.

Для простоты на рис. 9-9 не изображены программные модули MacApp; это сделало бы диаграмму излишне сложной, и, кроме того, диаграмма предназначена прежде всего для того, чтобы продемонстрировать проблемно-зависимую модульную архитектуру системы. Наша концепция состоит в том, что все классы и объекты MacApp являются примитивными абстракциями, видимыми в любой части приложения.

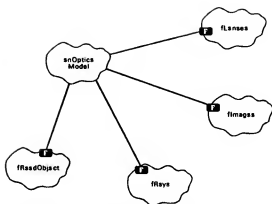


Рис. 9-10. Диаграмма объектов оптического эксперимента.

Отметим также, что мы решили сделать каждый программный модуль более низкого уровня видимым для интерфейса программного модуля следующего, более высокого уровня. Итак, UOpticsDialogs должен использовать UOpticsModels, UOpticsControllers должен использовать UOpticsDialogs и UOpticsModels и т.д. В конце концов основная программа MOptics должна использовать все шесть программных модулей более низкого уровня. Мы могли бы уменьшить эти зависимости (хотя и не очень сильно), так как интерфейс каждого программного модуля более высокого уровня должен видеть почти каждый программный модуль под собой. Мы решили не оптимизировать эти зависимости, потому что это сделало бы реализацию менее пригодной для понимания, так как ее модульная архитектура стала бы неправильной. В любом случае устранение одной или двух сторонних зависимостей не окажет сильного влияния на время перескомпиляции. Фактически в ходе создания нашей реализации интерфейс каждого программного модуля оставался относительно стабильным, и большая часть изменений происходила в реализации программного модуля. Таким образом, это уменьшало перескомпиляцию благодаря устранению старых программных модулей.

Почему же мы создали модульную архитектуру нашего приложения до разработки структуры его классов и объектов? На практике нам иногда казалось полезным сначала набросать в черновом виде модульную архитектуру, потому что это давало нам основную конструкцию, в рамках которой следовало постепенно разрабатывать наши проблемно-зависимые классы и объекты. Для небольших и средних по размеру систем, таких, как инструментальное средство разработки конструкций геометрической оптики, проектирование с программными модулями или пакетами обеспечивает приемлемую основную конструкцию. Для больших систем, таких, как система, описанная в гл. 12, мы должны проектировать еще и с подсистемами более высокого уровня. В каждом случае эти модули формируют программные модули для управления конфигурацией и для контроля версий, а также устанавливают пределы, в которых мы задаем работу разработчикам.

Основной довод, который склоняет к тому, чтобы в первую очередь разрабатывать модульную архитектуру, состоит в том, что это способствует ранней и возрастающей интеграции. Итак, первое, что мы могли бы сде-

лать, создать эти программные модули (каждый из них первоначально пустой) и затем компилировать, компоновать и выполнять результирующую пустую программу. Таким путем мы раньше начинаем применять наши инструментальные программные средства и обеспечиваем их соответствие данной архитектуре. Далее, по мере того как мы проектируем классы и объекты нашего приложения, мы можем сначала проверять их за пределами основной конструкции, а затем добавлять их к основной конструкции, которая, как нам уже известно, стабильна и функциональна. Таким образом, каждая рабочая версия нашего приложения обрывает функционально во времени контролируемые и предсказуемые способами.

Структура объектов

В проекте системы домашнего отопления (гл. 12) все проблемно-зависимые объекты были статичны. Это типично лишь для некоторых проблемных областей, но, конечно, не характерно для подавляющего большинства проблем автоматизации, с которыми мы встретимся. Например, проект инструментального средства разработки конструкций геометрической оптики включает только один явно статичный объект — сам объект-приложение. В любое время каждый объект-приложение может содержать ряд документов, каждый из которых представляет отдельный оптический эксперимент, а каждый оптический эксперимент может включать произвольное число линз. Кроме того, так как это приложение имеет однородный интерфейс, различные объекты-команды, объекты отображения и диалоговые объекты могут быть созданы и уничтожены в цикле существования данного приложения. Так же как мы поступали при описании ключевого механизма MacApp, мы можем использовать диаграммы объектов, чтобы представить ключевые механизмы среди транзитных объектов, которые формируют наш проект инструментального средства разработки конструкций геометрической оптики.

Из требований мы знаем, что оптический эксперимент состоит из набора линз, расположенных вдоль оптической скамьи. Если мы проведем основные лучи света от какого-либо реального объекта через этот набор линз, результирующий путь луча может сформировать изображения (и реальные, и мнимые). Это описывает состояние оптического эксперимента. Если обратиться к терминам объектно-ориентированного программирования, свойства каждого оптического эксперимента включают набор линз, набор изображений, набор лучей и один реальный объект, причем точные значения этих свойств могут изменяться со временем по мере того, как пользователь взаимодействует с приложением — добавляет, удаляет и передвигает отдельные линзы. На рис. 9-10 на диаграмме объектов изображено данное проектное решение о структуре оптического эксперимента.

Как показано на рис. 9-11, отдельные оптические эксперименты не существуют изолированно, они взаимодействуют с элементами ключевых механизмов MacApp: с документами, объектами отображения и командами. Здесь мы видим, что документ состоит из оптического эксперимента и объекта отображения. Это — приложение механизма документов MacApp. Используя механизм изображения MacApp, объект отображения и оптический эксперимент действуют вместе для поддержания соответствия между видимым изображением эксперимента и текущим состоянием эксперимента. Используя механизм команд в MacApp, объект отображения также может создать команды-посредники, которые различными способами изменяют состояние эксперимента.

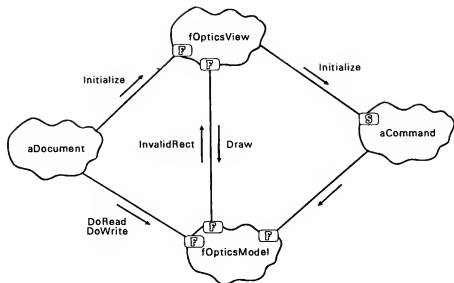


Рис. 9-11. Диаграмма объектов инструментального средства разработки конструкций геометрической оптики.

Запомним, что диаграмма объектов просто показывает прототипы действий. Следовательно, независимо от того, содержит ли данный оптический эксперимент одну линзу или сто линз, а текущий объект-команда представляет перемещение, выбор картинки сервисной программы, удаление, копирование, чистку или вставку, применяются такие же связи и взаимодействия, как изображенные на рис. 9-11.

Структура классов

Определение ключевых абстракций. В ходе проектирования инструментального средства разработки конструкций геометрической оптики мы обнаруживаем ряд новых объектов, имеющих сходные свойства, но все же непохожих друг на друга. Например, в требованиях упомянуты три типа рассеивающих линз и три типа собирающих линз. Эти шесть объектов, несомненно, являются линзами, но они имеют различия в форме, а рассеивающие и собирающие линзы отличаются способом преломления лучей. Поскольку мы можем выделить группы объектов определенного вида, подобно этим, следовательно между абстракциями, существующими в нашей проблемной области, следует установить отношения наследования. Если этого не сделать, наш проект будет намного хуже: в конце концов мы напишем больше исходных текстов, так как не выделили общие черты абстракций, и, кроме того, наш проект трудно будет расширять и поддерживать. Здесь мы имеем дело с системой, отличной от домашнего отопления, в которой вопросы наследования не выдвигались на первый план.

Большая часть структуры классов в оптическом эксперименте определяется интуитивно: есть класс `TLens` и оба класса `TConvergingLens` и `TDivergingLens` наследуют от этого базового класса. Кроме того, имеются еще классы, представляющие типы собирающих линз, и, следовательно, они наследуют от класса `TConvergingLens`:

- * `TDoubleConvexLens`
- * `TPlanoConvexLens`
- * `TConvexMeniscusLens`

Следующие классы представляют типы рассеивающих линз и, следовательно, наследуют от класса `TDivergingLens`:

- * `TDoubleConcaveLens`
- * `TPlanoConcaveLens`
- * `TConcaveMeniscusLens`

Изображения (реальные и мнимые) и реальные объекты также связаны между собой. В оптическом эксперименте есть оба типа объектов, но единственный реальный объект статичен в данном эксперименте, а все остальные объекты, как предполагается, динамические. Поскольку в нашем представлении о мире достаточно много различных понятий, мы решили ввести два класса `TRealOrVirtualImage` и `TRealObject`, которым присуще одинаковое поведение как подклассам класса `TImage`.

Достаточно ли существенны различия между реальными и мнимыми изображениями, чтобы служить основанием для ввода двух отдельных классов? В соответствии с требованиями единственное различие между этими двумя типами объектов состоит в том, что реальные изображения должны быть выполнены в темно-сером цвете, а мнимые — в светло-сером. Рассуждая таким образом, введение этих двух классов можно было бы считать оправданным. Однако то, что изображение является реальным или мнимым, зависит всего лишь от того, с какой стороны линзы появляется это изображение по отношению к первоначальному изображению.

Итак, мы могли бы использовать состояние, внутреннее для изображения (расположение изображения по отношению к линзе), для того, чтобы решить в каком цвете — темно- или светло-сером представить его на экране. В данном случае одного класса было бы не вполне достаточно. Какой же проект лучше? Мы считаем, что оба имеют свои достоинства. Тем не менее, поскольку мы не можем найти достаточно веских причин для создания двух классов, у нас остается один лишь класс `TRealOrVirtualImage`.

Следует обратить внимание еще на некоторые общие моменты. Есть ли какое-либо сходство между линзами и изображениями? Да, есть: и линзы, и изображения могут быть изображены на экране в объекте отображения, следовательно, есть поля и операции, являющиеся общими для этих двух больших классов объектов. Исходя из этого, мы можем ввести класс `TShape`, который будет суперклассом по отношению к обоим классам `TLens` и `TImage`. В наших требованиях речь идет о лучах света, и поэтому мы можем ввести класс `TRays`, который отражает поведение, общее для всех лучей. Так как лучи, подобно линзам и изображениям, могут быть изображены на экране в объекте отображения, мы также можем сделать класс `TRays` подклассом класса `TShape`. Как мы говорили в гл. 3, если объект состоит из других объектов, то его класс должен использовать классы его компонентов. Это относится к классу `TOpticsModel`, который инкапсулирует состояния, общие для всех оптических экспериментов, и поэтому использует классы `TLens`, `TImage` и `TRays`.

Оптическая скамья тоже считается объектом, так как является четко определенной абстракцией. Хотя она не участвует в прохождении лучей, она является действительной и в соответствии с требованиями по команде пользователя может быть видимой или невидимой. Поэтому мы вводим класс `TOpticsBench` как атрибут объекта отображения, который изображает на экране оптический эксперимент. На рис. 9-12 изображены почти все введенные нами классы. Для простоты мы не стали включать в схему используемые отношения для класса `TOpticsModel`. Отметим, что мы представили несколько классов (таких, как `TLens`) в виде абстрактных классов, так как они являются обобщенными классами, которые требуют дальнейшей специализации, прежде чем могут быть созданы какие-либо экземпляры этих классов.

Нужно также сказать, что мы представили данную часть проекта в более простом виде, чем это есть в действительности. На рис. 9-12 проект классов представлен в своем окончательном варианте, но не в процессе своего развития. Сначала мы спроектировали шесть типов линз в качестве непосредственных подклассов класса `TLens`. В ходе проектирования каждого класса мы пришли к тому, что этот процесс можно представить в виде некоторой схемы. Три собирающие и три рассеивающие линзы обнаружили достаточно сходства в своем представлении и поведении, и поэтому мы ввели два промежуточных класса `TConvergingClassLens` и `TDivergingClassLens`, которые отражают сходные представление и поведение каждого типа линз. В результате это упростило проектирование каждого из наиболее специализированных классов. Поскольку некое общее понятие объединяет линзы, реальный объект, лучи и изображения, мы ввели класс `TShape`. Тем самым мы наделили этот более абстрактный класс общим для данных подклассов представлением и поведением, и это снова позволило нам уменьшить объем исходных текстов.

Данный тип реорганизации классов довольно широко используется в процессе разработки. Конечно, мы не можем поступать так постоянно, потому что нам все время приходилось бы производить изменения в интерфейсе и большую часть времени заниматься перескомпиляцией программных модулей. Если дана большая система, каждому разработчику пришлось бы ждать, пока работа других разработчиков не стабилизируется, и в конечном счете замедлилась бы вся работа над проектом. Хотя на практике структура классов в системе некоторое время действительно остается неустойчивой, но по мере того как разрабатываются ключевые решения об архитектуре модулей, она стабилизируется. После этой стадии, на которой разработчики обычно приступают к написанию исходных текстов для интерфейсов и, следовательно, стабильность становится для них необходимым условием, единственные изменения, которые вносятся в структуру классов, это введение новых подклассов или добавление операций к интерфейсам существующих классов. Именно так мы поступали в проекте инструментального средства разработки конструкций геометрической оптики. Мы пришли к выводу, что следует планировать внесение изменений, потому что избежать их невозможно. Если на начальной стадии разработки проекта не вносить в него изменения, вряд ли можно будет быстро сделать его оптимальным. Может наступить момент, когда проявятся последствия плохого проектирования на начальном этапе, и вы не сможете двигаться дальше, и у вас не будет иного выбора, кроме как исправлять свои ошибки в проекте. Это потребует значительно большего времени, чем если бы вы решили эту проблему раньше.

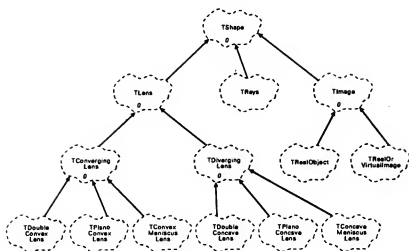


Рис. 9-12. Диаграмма классов оптического эксперимента.

Теперь мы можем приступить к проектированию интерфейса каждого из наших классов. Мы могли бы написать шаблоны для каждого класса, но поскольку Object Pascal — весьма выразительный язык, лучше написать их непосредственно на языке реализации. Однако следует быть осторожным и избегать непродуманных решений о представлении классов. Как мы говорили выше, представление класса должно обычно вытекать из его ожидаемого поведения. Итак, мы можем регулировать представление классов, чтобы сделать его оптимальным с учетом использования классов в будущем, причем поступать иначе крайне нежелательно, так как объекты-пользователи начинают зависеть от непродуманного представления классов, которое в процессе работы может измениться.

Проект оптической модели. Класс TShape представляет общее для всех классов поведение, которое может быть изображено в объекте отображения. Это означает, что он должен включать знания о механизме изображения в MacApp и знать такое понятие, как отмена объекта отображения. Мы могли бы начать создавать интерфейс к этому классу следующим образом:

```

TShape = object(TObject)
...
procedure TShape.IShape (AnOpticsModel : TOpticsModel);
procedure TShape.Invalidate;

function TShape.Frame : Rect;

end;

```

Многоточие показывает, где мы должны в конечном счете объявить поля этого класса, которые мы учитываем, так как это предусмотрено реализацией.

Заметим, что мы объявили метод `IShape`, который, как принято, мы должны вызвать сразу же после создания нового объекта класса. Обычно мы производим эти операции в `Object Pascal`, так как в отличие от `C++`, `Object Pascal` не позволяет нам объявлять операции конструктор или деструктор, а также потому, что `Object Pascal` в отличие от `Smalltalk` не поддерживает метаклассы. Мы также объявили процедуру `Invalidate` для поддержки механизма изображения в `MacApp`. Эта процедура использует общедоступную функцию `Frame`, возвращающую прямоугольник, в который заключен объект. Мы указали на эту функцию, потому что вскоре мы обратимся к ней для поддержки механизма действия мыши в `MacApp`.

Мы не экспортировали никаких методов, реализующих изображение, потому что, как мы увидим ниже, каждый из подклассов класса `TShape`, требует набора параметров, несколько отличающегося от набора параметров его собственного метода `Draw`. Поскольку для `Object Pascal` нежелательно слишком большое количество имен методов, мы не можем объявить эти различные методы, реализующие изображение, в одном месте.

Класс `TLens` немного сложнее, потому что он должен отразить поведение, общее для всех линз. Его интерфейс выглядит следующим образом:

`TLens = object(TShape)`

```
...
procedure TLens.IShape (AnOpticsModel : TOpticsModel); override;
procedure TLens.SetFocalLength (NewFocalLength : FocalLength);
procedure TLens.Select;
procedure TLens.Deselect;
procedure TLens.Invalidate; override;
procedure TLens.Highlight (FromHL, ToHL : HLState);
procedure TLens.DrawLens (DrawPosition: Boolean);
procedure TLens.Draw (Area : Rect; DrawPosition : Boolean);
procedure TLens.DoRead (ARefNum : Integer);
procedure TLens.DoWrite (ARefNum : Integer);

function TLens.DiskSpace : LongInt;
function TLens.Frame : Rect; override;
```

end;

Этот класс экспортирует процедуру `IShape`, которая замещает соответствующую процедуру класса `TShape`. Процедура `SetFocalLength` предназначена для установки фокусного расстояния. Но почему бы нам просто не объявить некоторое поле `fFocalLength`, к которому объект-пользователь может получить непосредственный доступ? Это привело бы нас к объектам-пользователям, находящимся в зависимости именно от данной конкретной реализации этого класса, но не от его абстрактных свойств. Изменение фокусного расстояния линзы имеет побочный эффект, кроме того, что просто дает новое значение некоторому полю: изменение фокусного расстояния линзы позволяет заново рассчитывать пути лучей. Обращение к этому методу дает новую оптическую модель, если для линзы задается новое фокусное расстояние.

Процедуры `Select` и `Deselect` представляют примитивные операции над всеми линзами, они необходимы для поддержки механизма действия мыши в `MacApp`. Например, пользователь может нажатием на клавишу мыши оперировать линзой, изображенной в объекте отображения; при этом сразу же вызывается метод `Select` для выбора этой линзы. Методы `Invalidate`, `Highlight`, `DrawLens` и `Draw` предназначены для поддержки механизма изо-

бражения в MacApp. Мы представляем метод DrawLens, являющийся примитивной операцией, которая всегда изображает линзу (и ее положение на оптической скамье, если оптическая скамья видима) и составную операцию Draw, которая вызывает метод DrawLens при условии, что рамка линзы попадает в отменяемую часть объекта отображения.

Методы DoRead, DoWrite и DiskSpace предназначены для поддержки механизма документов в MacApp. DiskSpace определяет объем памяти, необходимый для сохранения состояния объекта, а DoRead и DoWrite выполняют работу по восстановлению и сохранению состояния.

Интерфейсы к классам TConvergingLens и TDivergingLens — довольно простые. В каждом из них мы должны заменить метод IShape так, чтобы состояние каждого типа объектов устанавливалось соответствующим образом. Итак, мы могли бы записать:

```
TConvergingLens = object(TLens)
```

```
    procedure TConvergingLens.IShape (AnOpticsModel : TOpticsModel); override;
```

```
end;
```

Аналогично интерфейс каждого из шести наиболее специализированных классов линз должен всего лишь заменить метод IShape своего суперкласса. Например, мы могли бы записать интерфейс к классу TDoubleConvexLens следующим образом.

```
TDoubleConvexLens = object(TConvergingLens)
```

```
    procedure TDoubleConvexLens.IShape (AnOpticsModel : TOpticsModel); override;
```

```
end;
```

Может показаться странным, что у нас так много небольших классов, таких как TConvergingLens и TDoubleConvexLens. Следует еще раз повторить наши соображения по этому поводу: построение простых классов, включающих только конкретное поведение, имеет очень большое значение. Общее поведение может перейти к обобщенным классам, а это означает, что придется записывать и поддерживать меньше исходных текстов, что приводит к большей открытости системы и упрощает ее обслуживание. Кроме того, очень важно, что мы стараемся вводить классы, использующие терминологию, принятую в данной проблемной области, чтобы получить приложение, доступное для понимания. Введение классов, подобных TConvergingLens и TDoubleConvexLens, — важный этап в решении проблемы: создавая их, мы отражаем наше действительное представление о внешнем мире.

Рассмотрим класс TRays. Экземпляр данного класса представляет все основные лучи, проходящие через отдельную линзу. Достаточно ли прост этот класс? Почему вместо него мы не ввели класс TPrincipleRays, имеющий подклассы для каждого из трех основных лучей? Мы могли бы так сделать, но это усложнило бы проблему, по крайней мере в данном конкретном случае. Как мы выяснили, основные лучи, проходящие через отдельную линзу, внутренне взаимосвязаны: они исходят от одного и того же реального объекта, реального или мнимого изображения, и все они сходятся на одном и том же реальном или мнимом изображении. Более того, когда

мы вычислим точку, в которой сходятся два из этих трех лучей, мы можем легко определить путь третьего луча. Если бы мы рассматривали эти три луча по отдельности, обработка заняла бы у нас на треть больше времени, чем необходимо. Поэтому мы решили определить класс TRays как класс, включающий поведение всех трех основных лучей, проходящих через линзу.

Этот класс требует интерфейс, аналогичный интерфейсу класса TLens: требуются такие операции, как IShape, DrawRays, Draw и Frame для поддержки механизма изображения в MacApp. Поскольку семантика класса TRays представляет три основных луча, мы должны ввести метод, который устанавливает состояние объектов-лучей, так чтобы DrawRays и Draw могли нормально работать. Согласно требованиям, главный луч изображается проходящим от реального объекта (или реального либо мнимого изображения) через линзу. Луч, идущий от линзы, может быть затем проведен в обратном направлении — либо к реальному объекту (если он фокусируется на противоположной от первоначального изображения стороне линзы) или к мнимому изображению (если он фокусируется на той же стороне линзы, где находится первоначальное изображение). Итак, отдельный луч может быть определен двумя расположенными рядом отрезками, конечные точки которых находятся на первоначальном изображении, где-то на прямой, перпендикулярной оптической оси и на полученном реальном или мнимом изображении. Следовательно, все три основных луча пересекутся в полученном реальном или мнимом изображении при условии, что они фокусируются в точке, отличной от бесконечности, и мы исключаем помехи, вызванные сферической аберрацией.

Если даны только реальный объект или реальное либо мнимое изображение и линза, через которую проходит луч, то можно определить примитивную операцию для класса TRays, которая вычисляет точку, где пересекаются основные лучи. Полученная точка вместе с первоначальным изображением и линзой — достаточное условие, чтобы правильно изобразить все три основных луча. Поэтому мы будем экспортировать метод SetPoint, который выполняет это действие, несмотря на то что мы можем установить, как работает SetPoint. Как показано ниже, мы должны вернуться к методам структурного проектирования для дальнейшей декомпозиции этого метода, и, поступая таким образом, мы находим дополнительные классы, имеющие прямое отношение к нашему проекту.

Если принять эти проектные решения, можно записать интерфейс к классу TRays следующим образом:

```
TRays = object(TShape)
...
procedure TRays.IShape (AnOpticsModel : TOpticsModel); override;
procedure TRays.SetPoint (FromImage : TImage; ThroughLens : TLens);
procedure TRays.DrawRays;
procedure TRays.Draw (Area : Rect);

function TRays.Frame : Rect; override;
function TRays.ImageAt : Point;

end;
```

Как всегда в таких случаях, когда бы мы ни определяли модификатор, такой, как SetPoint, мы также определяем селектор, такой, как ImageAt, который возвращает состояние, связанное с модификатором.

Интерфейс класса TImage во многом напоминает интерфейс класса TLens. Цель класса TImage — включить поведение, общее для всех реальных объектов и реальных или мнимых изображений. Итак, мы могли бы записать интерфейс класса TImage следующим образом:

```
TImage = object(TShape)
...
  procedure TImage.IShape (AnOpticsModel : TOpticsModel); override;
  procedure TImage.DrawImage;
  procedure TImage.Draw (Area : Rect);

  function TImage.Frame : Rect; override;
end;
```

Два подкласса класса TImage большей частью переопределяют эти операции. Итак, мы могли бы записать:

```
TRealObject = object(TImage)
...
  procedure TRealObject.DrawImage; override;
  procedure TRealObject.Draw (Area : Rect); override;
end;

TRealOrVirtualImage = object(TImage)
...
  procedure TRealImage.SetPosition (TheTop : Point; FromLens : TLens);
  procedure TRealImage.DrawImage; override;
  procedure TRealImage.Draw (Area : Rect); override;
end;
```

Класс TRealOrVirtualImage экспортирует метод SetPosition, целью которого является обнаружение местоположения изображения, относящегося к данной линзе. Если оно расположено с той же стороны линзы, что и реальный объект, это изображение будет мнимым. Если оно расположено на противоположной стороне, то это будет реальное изображение. Метод SetPosition определен именно здесь, так как он может быть реализован только при условии, что мы имеем доступ к глубинному представлению класса.

Чтобы завершить наше проектирование ключевых классов, участвующих в оптических экспериментах, мы должны определить интерфейс класса TOpticsModel, который инкапсулирует все объекты, относящиеся к эксперименту. Как и для класса TLens, мы должны обеспечить операции, которые поддерживают механизмы документов, изображения и действия мыши в MacApp. Итак, мы должны обратиться к таким методам, как IOpticsModel, SelectAll, Draw, DoRead и DiskSpace.

Для определения остальных операций, которые данный класс должен экспортировать, нам следует подумать о том поведении, которое предполагаем увидеть у всех объектов данного класса. Поскольку оптический эксперимент включает набор линз, мы должны ввести такие модификаторы, как AddLens и RemoveLens, и такие селекторы, как NumberOfLenses, и такие итераторы, как EachLens. Нужно ли нам также обращаться к операциям, позволяющим объекту-пользователю манипулировать лучами и изображениями в ходе эксперимента? Наш ответ на этот вопрос — нет. Объект-пользо-

ватель не сможет разумно обращаться с этими объектами, и поэтому мы решили поместить лучи и изображения в ту часть оптического эксперимента, которая относится к реализации.

Тем не менее оптический эксперимент содержит достаточно знаний и условий, чтобы рассчитать пути лучей, и поэтому мы решили экспортировать метод `TraceRays`. Этот метод является основным для всех проблемно-зависимых алгоритмических транзакций внутри самого приложения. Несмотря на важность этого метода, нам не стоит думать о том, как реализовать его, ибо его реализация связана с вопросами очень низкого уровня абстракций. К счастью, как мы увидим ниже, реализация этого метода довольно проста, так как использует уже разработанные нами для класса `TLens` и его подклассов подробные интерфейсы.

Соединив эти проектные решения, мы теперь можем записать интерфейс класса `TOpticsModel`:

```
TOpticsModel = object(TObject)

...
procedure TOpticsModel.IOpticsModel;
procedure TOpticsModel.Free; override;
procedure TOpticsModel.TraceRays;
procedure TOpticsModel.AddLens (ALens : TLens);
procedure TOpticsModel.RemoveLens (ALens: TLens);
procedure TOpticsModel.SelectAll;
procedure TOpticsModel.DeselectAll;
procedure TOpticsModel.Highlight (FromHL, ToHL : HLState);
procedure TOpticsModel.Draw (Area : Rect; DrawPosition : Boolean);
procedure TOpticsModel.DoRead (ARefNum : Integer);
procedure TOpticsModel.DoWrite (ARefNum : Integer);

function TOpticsModel.DiskSpace : LongInt;
function TOpticsModel.CurrentSize : Point;
function TOpticsModel.NumberOfLenses : Integer;
function TOpticsModel.NumberOfSelections : Integer;

procedure TOpticsModel.EachLens (procedure DoToLens (ALens : TLens));
function TOpticsModel.FirstLensThat (function TestLens
                                     (ALens : TLens) : Boolean) : TLens;

end;
```

Как мы обычно поступаем в таких случаях, для удобства пользователя мы подразделили операции на модификаторы, селекторы и итераторы.

Как только мы определили интерфейс класса `TOpticsModel`, мы можем подумать о представлении этого класса. Здесь очевидны две реализации: 1) использование единого гетерогенного списка линз, изображений и лучей, 2) использование различных списков. Мы убеждены, что использование гетерогенного списка приводит к более сложной и менее эффективной реализации некоторых методов. Например, удаление линзы в таком случае потребовало бы обхода по списку объектов, многие из которых заведомо не являются линзами. Кроме того, расчет пути луча в первую очередь предусматривает устранение всех имеющихся изображений и лучей, и затем — создание новых, но при устранении старых изображений и лучей возникает та же проблема, что и при удалении линз из гетерогенного списка. Поэтому мы считаем, что состояние оптического эксперимента — это объединение гомогенных объектов. Итак, мы могли бы записать его поля следующим образом:

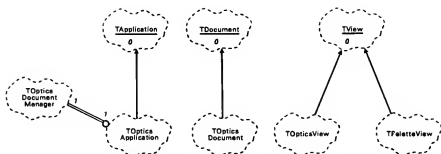


Рис. 9-13. Диаграмма классов приложения, документа и объекта отображения.

```

fView      : TView;
fLenses    : TLensList;
fImages    : TList;
fRays      : TList;
fRealObject : TrealObject;

```

Модель должна знать объект отображения, в котором она появляется (чтобы она могла устраивать линзы, изображения и лучи по мере их изменения), и таким образом мы обращаемся к полю `fView`.

Независимо от представления классов мы должны думать о чистке памяти, потому что эксперимент находится в динамическом состоянии, следовательно, мы экспортируем метод `Free`. Почему мы не экспортируем метод `Free` в каком-либо другом классе? Дело в том, что состояния объектов в классе `TOpticsModel` и состояния объектов в любом другом подклассе класса `TShape` существенно различаются. Объекты класса `TOpticsModel` инкапсулируют другие объекты, такие, как линзы, и поэтому, когда мы завершаем работу с отдельной оптической моделью (например, когда мы закрываем документ), мы должны также освободить подобъекты. С другой стороны, состояние таких объектов, как объекты класса `TLens`, весьма простое (оно состоит главным образом из скалярных величин), следовательно, они не содержат другие объекты, которые нужно высвобождать.

На этом мы завершаем наше проектирование ключевых классов, связанных с оптическим экспериментом. В соответствии с нашей модульной архитектурой системы каждый из рассмотренных нами классов описан в интерфейсе программного модуля `UOpticsModels`.

Проектирование классов интерфейса пользователя. В проектировании классов, составляющих интерфейс пользователя данного приложения, широко используется исследование для специализации существующих в `MacApp` классов. Их интерфейсы включают мало новых операций, наоборот, они в основном переопределяют операции, определенные в своих суперклассах. Программисту, впервые столкнувшемуся с `MacApp`, приходится методом проб и ошибок догадываться, какие операции отменять, но для разработчика, хорошо знакомого с механизмом `MacApp`, это не составляет никакого труда.

На рис. 9-13 представлена структура классов приложения, документов и объектов отображения инструментального средства разработки конструкций

геометрической оптики. Так как встроенные в MacApp классы TApplication, TDocument и TView важны для этого проекта, мы включили их в эту диаграмму. Их имена подчеркнуты для того, чтобы отметить, что они взяты извне (не из нашего приложения).

Как показано на диаграмме, мы ввели классы TOpticsApplication и TOpticsDocument, которые соответственно наследуют от встроенных классов TApplication и TDocument. Аналогично мы создали классы TOpticsView и TPaletteView как подклассы класса TView. Оставшийся класс TOpticsDocumentManager введен в соответствии с данными требованиями: наше приложение должно поддерживать не более четырех документов одновременно. Поскольку обработка, необходимая для перехода от одного эксперимента к другому, логически автономна, хотя чем-то и отличается от обычной обработки документов, имеющейся в MacApp, мы решили определить класс, отвечающий за обеспечение данного поведения. Включение знаний об управлении составными документами в отдельный класс упрощает структуру подкласса TOpticsApplication. Другой убедительный довод в пользу введения данного класса — это то, что требование обрабатывать не более четырех документов одновременно возможно может измениться. Экземпляры класса TOpticsDocumentManager должны отслеживать четыре различных документа. Со стороны мы видим, что данный класс должен знать, как добавить новый документ в приложение (когда пользователь выбирает New или Open в File-меню), как удалить документ из приложения (когда пользователь выбирает Close в File-меню) и как переименовать документ (когда пользователь выбирает Save As в File меню). Кроме того, данный класс должен знать, как активизировать окно, связанное с документом, когда пользователь выбирает имя данного документа из соответствующего меню. Эти четыре модификатора являются примитивными операциями, так как мы можем реализовать их только при условии, что мы имеем доступ к глубинной реализации класса, поэтому их следует включить в интерфейс TOpticsDocumentManager. С помощью селекторов данный класс содержит достаточно сведений, чтобы знать, можно ли открыть еще документы, помимо имеющихся, и вообще имеются ли открытые документы.

Теперь мы можем записать интерфейс класса TOpticsDocumentManager следующим образом:

```
TOpticsDocumentManager = object(TObject)
```

```
...
procedure TOpticsDocumentManager.IOpticsDocumentManager;
procedure TOpticsDocumentManager.DoSetUpMenus;
procedure TOpticsDocumentManager.AddDocument (ADocument : TOpticsDocument);
procedure TOpticsDocumentManager.DeleteDocument (ADocument : TOpticsDocument);
procedure TOpticsDocumentManager.RenameDocument (ADocument : TOpticsDocument);
procedure TOpticsDocumentManager.FocusOnADocument (ACmdNumber : CcmdNumber);

function TOpticsDocumentManager.CanAddDocument : Boolean;
function TOpticsDocumentManager.HasOpenDocuments : Boolean;
```

```
end;
```

Данный класс также экспортирует метод IOpticsDocumentManager для инициализации объектов и метод DoSetUpMenus, который необходим нам для поддержки механизма команд меню в MacApp.

Класс TOpticsApplication в основном просто переопределяет некоторые операции своего базового класса TApplication для поддержки проблемно-зави-

симой обработки меню и документов. Поскольку мы хотим, чтобы данный класс использовал средства объекта управления документами, который мы только что спроектировали, мы должны включить в класс `TOpticsApplication` поле, которое содержит объект класса `TOpticsDocumentManager`. Это поле представляет проблемно-зависимое состояние, и, следовательно, мы должны также включить в этот класс соответствующую инициализацию и методы очистки памяти. Итак, мы можем записать интерфейс класса `TOpticsApplication` следующим образом:

```
TOpticsApplication = object(TApplication)
```

```
    IOpticsDocumentManager : TOpticsDocumentManager;
```

```
    procedure TOpticsApplication.IOpticsApplication;
```

```
    procedure TOpticsApplication.Free; override;
```

```
    procedure TOpticsApplication.DoSetUpMenus; override;
```

```
    procedure TOpticsApplication.AddDocument (ANewDocument : TDocument); override;
```

```
    procedure TOpticsApplication.DeleteDocument (DocToDelete : TDocument); override;
```

```
    function TOpticsApplication.DoMenuCommand
```

```
        (ACmdNumber : CmdNumber) : TCommand; override;
```

```
    function TOpticsApplication.DoMakeDocument
```

```
        (ItsCmdNumber : CmdNumber) : TDocument; override;
```

```
    function TOpticsApplication.MakeViewForAlienClipboard : TView; override;
```

```
end;
```

Метод `MakeViewForAlienClipboard` предназначен для поддержки буфера Macintosh. Когда начинается работа с новым приложением, могут иметься объекты (оставшиеся из буфера другого приложения), которые обрабатывало инструментальное программное средство разработки конструкций геометрической оптики. Данный метод проверяет наличие таких объектов и создаст буфер для хранения любых найденных объектов.

Два класса `TOpticsDocumentManager` и `TOpticsApplication` объявлены в программном модуле `UOptics` согласно с ранее введенной модульной архитектурой.

Проект класса `TOpticsDocument` во многом аналогичен проекту класса `TOpticsApplication`. Его целью является переопределение методов суперкласса для того, чтобы поддержать проблемно-зависимую обработку MacApp команд меню и механизмов документов. Таким образом, мы можем написать интерфейс класса `TOpticsDocument` следующим образом:

```
TOpticsDocument = object(TDocument)
```

```
    ...
```

```
    procedure TOpticsDocument.IOpticsDocument;
```

```
    procedure TOpticsDocument.Free; override;
```

```
    procedure TOpticsDocument.DoMakeWindows; override;
```

```
    procedure TOpticsDocument.DoMakeViews (ForPrinting : Boolean); override;
```

```
    procedure TOpticsDocument.DoNeedDiskSpace (var DataForkBytes,
                                                RsrcForkBytes : LongInt); override;
```

```
    procedure TOpticsDocument.DoRead (ARefNum : Integer;
```

```
        RsrcExists,
```

```
        ForPrinting : Boolean); override;
```

```
    procedure TOpticsDocument.DoWrite (ARefNum : Integer;
```

```
        MakingCopy : Boolean); override;
```

```
end;
```

Данный класс объявлен в интерфейсе программного модуля `UOpticsDocuments`.

Рассмотрим более подробно классы `TOpticsView` и `TPaletteView`. Объекты этих двух классов совместно реализуют окно палитры подобно тому, как показано на рис. 9-3. Экземпляры класса `TOpticsView` выводят на экран изображения элементов данного оптического эксперимента, тогда как экземпляры класса `TPaletteView` обеспечивают выбор сервисной программы, которую пользователь может применить для создания или модификации оптического эксперимента в соответствующем оптическом объекте отображения.

Большой частью интерфейс класса `TOpticsView` представляет собой просто переопределенные операции его суперкласса `TView`, для того чтобы поддержать механизмы изображения и команд меню `MacApp`. Поэтому мы должны заменить методы `CalcMinSize` (для поддержки объектов отображения с переменными размерами), `DoHighlightSelections`, `Draw`, `DoSetUpMenus`, `DoMouseCommand`, `DoMenuCommand` и `DoSetCursor`.

В соответствии с требованиями устанавливаем, что оптические объекты отображения могут содержать несколько визуальных атрибутов, включая метки шкалы, разделители страниц и оптическую скамью, и каждый из них может быть или не быть видимым согласно желанию пользователя. Эти атрибуты отражают сущность данной проблемной области, и поэтому имеют смысл экспортировать такие операции, как `ToggleDrawOpticalBench` и `ToggleDrawGridLines`, которые обеспечивают данное поведение. Мы решили экспортировать скорее операции, которые переключают соответствующее `Boolean` состояние, а не устанавливать `True`- или `False`-состояние для каждого атрибута. Этим достигается лучшее согласование с механизмом команд меню `MacApp`, в котором определенное меню просто переключается из одного состояния в другое. Тем не менее надо отдавать себе отчет в том, что эти операции не только устанавливают состояние некоторого булева поля. Изменение состояния одного из этих атрибутов также заставляет обновлять объект отображения. Должны ли мы также экспортировать операции селекторы, которые восстанавливают состояние этих свойств? Теоретически да, но так как `Object Pascal` не инкапсулирует свои поля и данное состояние является примитивным, мы разрешили объекту-пользователю прямой доступ к соответствующему состоянию. Это отличается от нашего проектного решения для селекторов, определенных в классе `TOpticsDocumentManager`, потому что в последнем случае мы ожидаем, что каждое возвращаемое значение должно было бы скорее вычислено, чем взято непосредственно в поле.

Что касается его представления, то каждый оптический объект отображения должен быть видимым для модели, которую он изображает на экране, для согласования с механизмом изображения `MacApp`. Поскольку мы имеем два отдельных класса `TOpticsModel` и `TOpticalBench`, то мы должны включить два поля, по одному для каждого объекта этих классов. Оптический объект отображения должен также быть видимым для его палитры, так как объект отображения знает, какую сервисную программу выбрал пользователь. Таким образом, мы должны включить поле, которое содержит объект-палитра. И наконец, мы должны включить булево поле для каждого атрибута, который может переключаться в объекте отображения.

Исходя из этих проектных решений и решений реализации, мы можем написать интерфейс класса `TOpticsView` следующим образом:


```

TOpticsView = object(TView)
  fOpticsModel      : TOpticsModel;
  fOpticalBench     : TOpticalBench;
  fPaletteView      : TPaletteView;
  fDrawOpticalBench : Boolean;
  fDrawGridLines    : Boolean;
  fDrawPageBreaks   : Boolean;
  fAutoGridEnabled  : Boolean;
  procedure TOpticsView.IOpticsView (ADocument : TDocument;
                                       APalette : TPaletteView;
                                       AModel : TOpticsModel);
  procedure TOpticsView.CalcMinSize (var MinSize : VPoint); override;
  procedure TOpticsView.ToggleDrawOpticalBench;
  procedure TOpticsView.ToggleDrawGridLines;
  procedure TOpticsView.ToggleDrawPageBreaks;
  procedure TOpticsView.ToggleAutoGridEnabled;
  procedure TOpticsView.DoHighlightSelection (FromHL, ToHL : HLState); override;
  procedure TOpticsView.DrawGridLine (Area : Rect);
  procedure TOpticsView.Draw (Area : Rect); override;
  procedure TOpticsView.DoSetUpMenus; override;
  function TOpticsView.DoMouseCommand (var TheMouse : Point;
                                       var Info : EventInfo;
                                       var Hysteresis : Point) : TCommand; override;
  function TOpticsView.DoMouseCommand (ACmdNumber : CmdNumber) : TCommand; override;
  function TOpticsView.DoSetCursor (LocalPoint : Point;
                                    CursorRgn : RgnHandle) : Boolean; override;
end;

```

Объект класса TPaletteView представляет палитру сервисных программ, среди которых пользователь может сделать выбор. Для нашего приложения соответствующее инструментальное средство включает указатель (для выбора линзы) и сервисную программу изображения для каждой из шести типов линз, которые мы должны моделировать. TPaletteView должен инкапсулировать знания о том, как изображать данные сервисные программы, а также как реагировать на выбор новой сервисной программы пользователем. В нашем проекте объект-палитра не знает, что делать с сервисной программой, а объект отображения знает. Поэтому наше проектное решение заключается в том, что, когда пользователь выбирает новую сервисную программу, объекту-палитре необходимо только отменить выбор старой сервисной программы и выделить новую сервисную программу. Изображаемый объект отображения, соответствующий данному объекту-палитре, может запросить объект-палитру о выбранной в настоящий момент сервисной программе (которая либо указатель, либо линза), когда это необходимо. Для поддержки этих решений мы должны ввести селекторы для класса TPaletteView, такие, как IsPointer (возвращающий True, если указательная сервисная программа выбрана в данный момент и False в другом случае) и CurrentTool (который возвращает объект подкласса TLens, соответствующий выбранной в данный момент сервисной программе изображения линзы). Поэтому мы можем записать интерфейс данного класса следующим образом:

```

TPaletteView = object(TView)
  ...
  procedure TPaletteView.IPaletteView (Document : TDocument;
                                       AModel : TOpticsModel);
  procedure TPaletteView.DoHighlightSelection (FromHL, ToHL : HLState);
  override;
  procedure TPaletteView.Draw (Area : Rect); override;

```

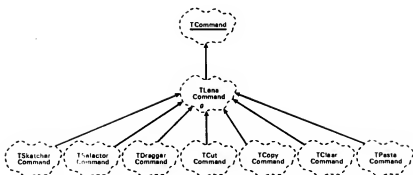


Рис. 9-14. Диаграмма классов оптических команд.

```

function TPaletteView.DoMouseCommand
    (var TheMouse : Point;
     var Info : EventInfo;
     var Hysteresis : Point) : TCommand;
    override;

function TPaletteView.IsPointer : Boolean;

function TPaletteView.CurrentTool : TLens;
  
```

end;

Отметим, что `CurrentTool` возвращает объект класса `TLens`. В действительности данная функция всегда возвращает объект некоторого подкласса класса `TLens`, такого, как класса `TPlanoConcaveLens`. Наш проект структуры классов для линз определяет общее поведение линз в классе `TLens`. Поэтому любой объект-пользователь, который имеет доступ к текущей сервисной программе палитры, может полностью манипулировать результирующим объектом-линзой, даже если класс объекта мог быть неизвестным до момента выполнения. В ситуации, подобной данной, полиморфизм и динамическая связь служат наилучшим образом. К счастью, мы получили выигрыш в статической проверке типов `Object Pascal`, потому что все результирующие объекты совместно используют общий класс-предок `TLens`. В соответствии с нашей модульной архитектурой классы `TOpticsView` и `TPaletteView` объявлены в интерфейсе программного модуля `UOpticsViews`.

Проектирование команд эксперимента. Проблемно-зависимые команды представляют собой другие классы существующие, для нашего проекта. Согласно требованиям, наше приложение должно поддерживать выбор новых линз вдоль оптической скамьи, выбор существующих линз и перемещение всех выбранных линз. Совместно с руководством интерфейса пользователя `Macintosh` наше приложение должно также поддерживать удаление, копирование, чистку и вставку линз внутри как данного оптического эксперимента, так и в других экспериментах. `MacApp` предусматривает класс `TCommand` как часть его механизма команд меню, и поэтому мы можем ввести подклассы данного класса для поддержки различных проблемно-зависимых команд, которые нам необходимы.

На рис. 9-14 представлены наши проектные решения по этим классам. На самом нижнем уровне иерархии находится семь специализированных классов, которые являются подклассами класса `TLensCommand`. Последний в свою очередь является подклассом класса `TCommand` в `MacApp`. Сначала мы сделали каждый из семи специализированных классов прямым наследником класса `TCommand`, но скоро мы обнаружили некий шаблон. Необходимо, чтобы каждый объект-команда имел поле, которое указывало бы на модель, на которую он воздействует, и для каждого объекта-команды необходимо совместно использовать общее поведение для ограниченного перемещения мыши. Поэтому мы ввели класс `TLensCommand` и перенесли все общее поведение из семи специализированных классов в данный промежуточный класс.

Мы можем записать интерфейс класса `TLensCommand`. Так как команда действует на линзы в оптическом эксперименте, мы должны включить поле, содержащее объект класса `TOpticsModel`. Поэтому, как нам станет очевидно позже, мы также включаем `Boolean`-поле, которое указывает, будет изображаться на экране положение линз, которыми манипулирует данный объект команда, вдоль оптической оси или нет.

Таким образом, мы можем записать

```
TLensCommand = object(TCommand)
fOpticsModel      : TOpticsModel;
fDrawPosition     : Boolean;

procedure TLensCommand.ILensCommand
(ASuperView : TView;
AnOpticsModel : TOpticsModel;
DrawPosition : Boolean);
procedure TLensCommand.TrackConstrain
(AnchorPoint,
PreviousPoint : VPoint;
var NextPoint : VPoint); override;
end;
```

Интерфейсы всех семи специализированных классов очень похожи. Для большей части интерфейса мы должны переопределить такие операции, как `Commit`, `DoIt`, `UndoIt` и `RedoIt`, которые обеспечивают специфическое поведение для каждого класса и необходимы для поддержки механизма команд меню `MacApp`. Каким образом эти методы вызываются? С помощью механизма главного цикла событий метод приложения `HandleEvent` первым реагирует на событие, извлекая метод `DispatchEvent`. Как только метод `DispatchEvent` выполнен, управление возвращается к `HandleEvent`, который затем извлекает метод приложения `PerformCommand` для обработки какой-либо команды, которую мог создать объект управления событием (например, через проблемно-зависимую реализацию метода `DoMouseCommand`). Метод `PerformCommand` выполняет наиболее важную часть работы, вызывая метод `DoIt` соответствующего объекта команды. Аналогично методы `UndoIt` и `RedoIt` вызываются объектом-приложением как часть его стандартных методов управления меню.

Сейчас мы можем записать интерфейс класса `TSketcherCommand` следующим образом:

```
TSketcherCommand = object (TLensCommand)
fLens : TLens;
```

```

procedure TSketcherCommand.iLensCommand (ASuperView : TView;
                                           AnOpticsModel : TOpticsModel;
                                           DrawPosition : Boolean); override;

procedure TSketcherCommand.Free; override;
procedure TSketcherCommand.Commit; override;
procedure TSketcherCommand.DoIt; override;
procedure TSketcherCommand.UndoIt; override;
procedure TSketcherCommand.RedoIt; override;
procedure TSketcherCommand.TrackFeedback (AnchorPoint,
                                           NextPoint : VPoint;
                                           TurnItOn,
                                           MouseDidMove : Boolean); override;

function TSketcherCommand.TrackMouse (ATrackPhase : TrackPhase;
                                       var AnchorPoint, PreviousPoint,
                                       NextPoint : VPoint;
                                       MouseDidMove : Boolean) : TCommand;
                                       override;

```

end;

Поскольку объект выбор картинки должен выбрать линзу определенного типа, то этот класс включает поле `fLens` для сохранения данного состояния. Как мы увидим ниже, объект отображения отвечает за создание объекта-команды «выбора картинки», затем устанавливает значение данного поля с помощью метода `CurrentTool` соответствующего объекта-палитры.

Мы не будем показывать интерфейсы остальных специализированных классов, поскольку они аналогичны рассмотренным выше. Все семь классов, так же как и класс `TLensCommand`, объявляются в интерфейсе программного модуля `UOpticsCommands`. Что представляет собой программный модуль `UOpticsDialogs`? Как правило, мы экспортировали неспециализированные интерфейсы для зависимых от приложения диалогов. Окна диалогов в общем случае конструируются на основе класса `TDialogView` в `MacApp`: диалог обычно управляет экземплярами таких классов, как `TPicture`, `TPopup`, `TButton`, `TEditText` и `TStaticText`. Только в нескольких случаях требуется ввести новые подклассы диалогов, и фактически все эти классы могут быть скрыты в реализации программного модуля диалога. Поэтому мы можем использовать на высоком уровне абстракции программный модуль `UOpticsDialogs` для экспортирования набора общедоступных программ, которые существуют только для вывода на экран диалогов и затем возврата значений, полученных в результате взаимодействия диалога с пользователем.

Какие диалоги нам необходимы? Помимо тех проектных решений, которые мы до сих пор рассматривали, существуют еще и другие проектные решения. Теперь нам необходимо посмотреть на все с позиции пользователя интерфейса. Для решения нашей проблемы требуется основанная на окнах система, которая позволяет выбирать меню и поддерживает работу с мышью; кроме этого, у нас мало ориентиров. На этой стадии нашего проектирования мы ввели ядро приложения, состоящее из его ключевых классов и самых существенных механизмов. Теперь мы должны рассмотреть проект интерфейса пользователя.

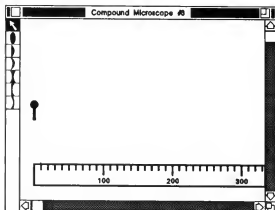


Рис. 9-15. Проект окна инструментального средства разработки конструкций геометрической оптики.

9.3. РАЗВИТИЕ

Реализация интерфейса пользователя

Искусство проектирования интерфейса пользователя ставит перед разработчиком бесчисленное количество возможностей. Например, должны ли мы использовать простой процессор командной строки, или возможно вместо него спускающееся меню (или и то и другое)? Должны ли мы предполагать, что нашими выходными устройствами являются только простые, символьные терминалы, или у нас будут дисплеи, которые позволят нам изобретать дружелюбный интерфейс, используя составные перекрывающиеся окна с графическими объектами отображения, которые могут быть помещены в подокна и снабжены звуковым сигналом? Что можно сказать о самих командах: должны ли мы еще использовать краткие непонятные сокращения, такие, как *ggr* или *mkfs*, или должны использовать длинные, значимые имена? Человеческий фактор, технические ограничения, исторические причины и личное предпочтение членов команды разработчиков — все эти обстоятельства делают задачу создания полезного, выразительного, непротиворечивого интерфейса человек-машина очень сложной. К счастью, стандартные интерфейсы пользователей, такие, как Motif [11], и программы, включенные в MacApp, в течение долгого времени помогают разработчикам строить различные по виду и наполнению приложения.

Как мы уже видели, мы можем отделить подробные проектные решения интерфейса пользователя от некоторых проектных решений программного обеспечения высокого уровня. Интерфейс пользователя, необходимый для решения некоторых проблем, часто является всего лишь косметической оболочкой, которая окружает ядро приложения, и поэтому вполне можно создать несколько различных видов интерфейсов для одной и той же системы. Для таких приложений, как инструментальное средство разработки конструкций геометрической оптики и различные программы изображения и рисования, интерфейс пользователя является больше, чем просто фасад; изящный и выразительный интерфейс пользователя существенно полезен для таких продуктов. К счастью, прототипируя интерфейс пользователя, до того как мы пол-

ностью реализуем или даже спроектируем все необходимые механизмы, мы можем поддерживать обратную связь с возможными пользователями. Своевременная обратная связь с пользователями помогает настраивать семантику интерфейса и очищать его от идiosинкразических наростов, зазубрин и царапин.

Выбор MacApp для нашей реализации непосредственно ограничивает возможности интерфейса пользователя. Эти ограничения, так же как и наши требования, означают, что мы предполагаем использовать окна, спускающиеся меню и фактически целую гамму механизмов, описанных в стандарте интерфейса пользователя Macintosh. Таким образом, наша задача упрощается до проблемы, связанной с решением о внешнем виде различных окон и диалогов в нашем приложении, а также о выборе меню и команд.

Проектирование окон. На рис. 9-15 показан макет окна, который мы можем использовать для изображения на экране оптического эксперимента, используя палитру и просматриваемый объект отображения. Палитра содержит указатель и шесть картинок сервисных программ, по одной для каждого типа линз. Пользователь может нажать и отпустить клавишу мыши на указателе или на любой сервисной программе для того, чтобы выбрать ее. Мы указываем на то, что определенный элемент палитры выбран с отображением его изображения. Мы также изображаем просматриваемый объект отображения с наибольшим количеством атрибутов, включая оптическую скамью, единственный действительный объект, горизонтальные и вертикальные метки шкалы.

Проектирование меню. Хотя стандарт интерфейса пользователя Macintosh предполагает, что все меню должны появиться на линейке меню, тем не менее существуют причины, по которым каждое приложение должно содержать определенные стандартные меню. Первое меню, которое необходимо иметь, — Apple-меню, на которое указывает символ @. Первый элемент в данном меню используется для вывода окна диалога, содержащего меню приложения, права на копирование и для некоторых приложений вспомогательную информацию. В нашем приложении используется следующее имя для данного элемента:

* About Optics...

Многоточие указывает пользователю, что выбранный элемент меню выведет на экран окно диалога. Оставшиеся элементы Apple-меню обычно посвящены настольной добавочной информации.

Следующим стандартным меню является File-меню. Его цель — обеспечить простые команды для работы с файлами, такие, как Open, Close и Save. В него также включаются две команды вывода на печать и команда выхода из приложения. Мы включили следующие элементы в данное меню: New, Open, Close, Save, Save as..., Save to Copy..., Revert to Saved, Page Setup..., Print..., Quit.

Некоторые из этих элементов, такие, как New, Open и Save, имеют эквиваленты в виде командных клавиш клавиатуры (Command-N, Command-O и Command-S соответственно), определенных в стандарте интерфейса пользователя Macintosh. Некоторые элементы меню могут быть недоступными в определенные моменты времени. Например, элемент Close должен быть доступным, только когда существует по крайней мере один открытый документ, а элемент Save должен быть доступным, только когда существует открытый документ, который был недавно изменен. Так как требования к приложению предусматривают управление не более чем четырьмя открытыми документами одновременно, то мы имеем доступными New и Open только

тогда, когда существует меньше чем четыре открытых документа в настоящий момент. Для того чтобы была более тесная обратная связь в случае, когда пользователь создает или открывает документ, мы решили ввести диалог, который предупреждает пользователя, пытающегося открыть более четырех документов. Третье стандартное меню Edit обычно включает команды удалить, передвинуть и скопировать объекты. Мы включаем следующие элементы в данное меню: Undo, Cut, Copy, Paste, Clear, Select All, Show Clipboard.

Большинство из этих элементов имеют стандартные эквиваленты в виде командных клавиш клавиатуры. Когда пользователь вырезает или копирует выбранные элементы, они помещаются в специальный объект отображения, называемый буфером вырезанного изображения. Элементы могут быть помещены из буфера вырезанного изображения обратно в эксперимент. Так как содержание буфера вырезанного изображения продолжает существовать после выхода из приложения, пользователь может вырезать и вставить элементы среди приложений, предполагая, что приложения соответствуют значениям элементов. Команда Show Clipboard изображает на экране буфер вырезанного изображения объекта отображения, так что можно видеть недавно вырезанные или скопированные объекты.

Рассмотрим теперь нестандартное меню. Мы изобретаем эти меню и их элементы, задаваясь вопросом, какие операции пользователь может производить над приложением как над единым целым. Не все такие операции в отличие от элементов меню являются удобными. Например, выбор, перемещение и выбор картинки сервисной программы лучше выполняются с помощью действий мыши. Тем не менее существует несколько типов операций, не связанных с мышью: команды для изображения на экране различных атрибутов объектов отображения, команды для фокусирования на одном из четырех открытых документов и команды для изображения свойств выбранных линз. Мы размещаем эти команды в одном из двух меню. Первое меню, называемое View, содержит следующие элементы:

- * Drawing Size...
- * Show Optical Bench
- * Show Grid Lines
- * Show Page Breaks
- * Enable Autogrid

Выбор первого элемента в данном меню выводит на экран окно диалога, с помощью которого пользователь может регулировать размер объекта отображения, округляя его до размера ближайшей наибольшей страницы. Оставшиеся четыре элемента меню являются командами переключения, которые отражают состояние отдельных атрибутов. Например, когда оптическая скамья невидима в настоящий момент, второй элемент меню читается, как указано выше. Когда этот элемент выбран, оптическая скамья изображается и данный элемент изменяется на Hide Optical Bench.

Оставшиеся три элемента меню имеют аналогичное поведение. Мы вызываем следующее меню Experiments и оно содержит следующие элементы:

- * <<...>>
- * <<...>>
- * <<...>>
- * <<...>>
- * Lens Specifications...

Первые четыре элемента представляют имена каждого из четырех документов. Всякий раз когда существующий эксперимент открывается, мы хотим, чтобы его имя появилось как элемент в данном меню. Если это новый документ, имя ему дается приложением (в форме Untitled n, где n — монотонно возрастающее число, установленное приложением). Когда приложение закрывается, его имя убирается из меню, и другие имена перемещаются вверх в списке. Мы должны также не забыть переименовать документ, когда пользователь выбирает элемент Save as... из File-меню. Когда пользователь выбирает элемент из Experiment-меню, указывающий на отдельный документ, мы хотим, чтобы приложение вывело соответствующее окно как внешнее, так чтобы пользователь мог легко найти эксперимент среди всех окон, которые могут загромождать экран.

Последний элемент в данном меню LensSpecifications.. является доступным, только когда выбрана одна единственная линза. Когда пользователь выбирает данный элемент, появляется окно диалога, которое показывает тип линзы, ее положение вдоль оптической скамьи и ее фокусное расстояние. Мы позволим пользователю редактировать только фокусное расстояние линзы, как это определяется нашими требованиями.

Мы уже говорили выше, что такие ресурсы, как меню, лучше определены в ресурсном файле, поэтому мы можем изменить внешний вид приложения без изменения его исходного текста. Например, мы можем отразить наши решения проектирования, касающиеся содержания View-меню включив следующее в наш ресурсный файл:

```
resource 'omnu' (4) {
    4,
    nextMenuProc,
    AllItems & ~ (MenuItem2 | MenuItem6),
    enabled,
    "View",
    /* {1} */ "Drawing Size...", noIcon, " ", " ", plain, 1001;
    /* {2} */ " ", noIcon, " ", " ", plain, noCommand;
    /* {3} */ "Show Optical Bench", noIcon, " ", " ", plain, 1002;
    /* {4} */ "Show Grid Lines", noIcon, " ", " ", plain, 1003;
    /* {5} */ "Show Page Breaks", noIcon, " ", " ", plain, 1004;
    /* {6} */ " ", noIcon, " ", " ", plain, noCommand;
    /* {7} */ "Enable Autogrid", noIcon, " ", " ", plain, 1005;
};
```

Проектирование диалога. Четыре связанных друг с другом диалога соответствуют меню AboutOptics...; диалог предупреждения для использования, когда четыре документа уже открыты и пользователь старается создать или открыть еще один документ; Drawing Size... диалог; Lens Specification... диалог. Независимо, что мы реализуем эти диалоги сейчас, но поскольку мы хотим, чтобы они имели единообразный вид и наполнение, важно проектировать их в одном и том же стиле. Поэтому в нашей модульной архитектуре мы поместили все диалоги в программный модуль UOpticsDialogs: размещаясь в одном программном модуле, они могут совместно использовать все принципы реализации.

На рис. 9-16 показаны макеты двух наиболее важных диалогов. Диалог Drawing Size... является достаточно сложным. Пользователь может выбрать направление, в котором нумеруются страницы (с помощью двух радиоклавиш), так же как размер страницы (нажимая и отпуская клавишу мыши на сетке, которая в свою очередь обновляет сетку, выделяя и высоту, и

ширину статических текстовых элементов). Диалог сложен для реализации, хотя пользователь интуитивно это понимает, отчасти из-за того, что сетка может изменяться в зависимости от типа выходного устройства и направления вывода на печать, увеличения и размера бумаги выбранного пользователем. Второй диалог, используемый в команде *Lens Specifications*, намного проще; он только позволяет пользователю редактировать фокусное расстояние, показанное в диалоге. Теперь мы можем создать интерфейс для этих диалогов:

```
unit UOpticsDialogs; interface
uses
    UMacApp, UPrinting, UDialog, ToolUtils, Packages,
    UOpticsModels;

{$S UDial}
procedure InitializeOpticsDialogs;
procedure DisplayAboutOpticsDialog;
procedure DisplayNoMoreDocumentsDialog;
procedure DisplayDrawingSizeDialog (CurrentModelSize : Point;
    CurrentResolution : Point;
    CurrentPageSize : VPoint;
    var CurrentViewSize : VPoint;
    var CurrentDirection : VhSelect;
    var AcceptChanges : Boolean);
procedure DisplayLensSpecificationDialog (SelectedLens : TLens;
    var NewFocalLength : FocalLength;
    var AcceptChanges : Boolean);

implementation
    {$I UOpticsDialogs.Incl.p}
end.
```

Для полноты мы должны обеспечить полный интерфейс с данным модулем. Отметим использование предложений: мы доказываем зависимость от такого модуля других модулей, которые являются частью *MacApp* (такие, как *UMacApp*, *UPrinting* и *UDialog*) и частью исходного кода нашего приложения (такое, как *UOpticsModels*). Отметим также директиву компилятора (*\$\$UDial*), которая указывает, что весь скомпилированный код помещается в сегменте, называемом *UDial*. Наконец, общедоступная процедура *InitializeOpticsDialogs* существует для того, чтобы инициализировать состояние определенного модуля (такого как модуль диалога *MacApp*).

Реализация классов интерфейса пользователя. На данном этапе мы можем реализовать все интерфейсные модули, которые спроектировали. Так как эта книга посвящена проектированию, а не кодированию, то мы не обеспечиваем здесь полную реализацию данной программы. Кроме того, конечная реализация состоит из более 3000 строк *Object Pascal* и почти 2000 строк исходного текста в ресурсном файле (с другой стороны, можете ли вы представить проектирование без использования богатой библиотеки классов, такой, как *MacApp*). Тем не менее существуют еще разделы проектирования, которые заслуживают обсуждения. В частности, мы должны показать, как наше проектирование строится на основе механизмов *MacApp*. Мы также хотим продемонстрировать, как определенное представление решений естественно следует решениям интерфейса. В конце мы хотим показать, как при использовании методов объектно-ориентированного проектирования проект системы развивается и реализуется.

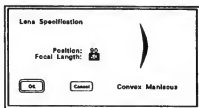
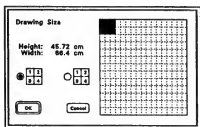


Рис. 9-16. Диалоги инструментального средства разработки конструкций геометрической оптики.

Реализацию можно начинать с любого тела модуля, но мы начинаем с классов, которые обеспечивают каркас приложения и интерфейс пользователя, поэтому мы можем непосредственно получить полностью выполняемые системы. Поэтому начнем с класса `TOpticDocumentManager`.

Выше мы проектировали меню и элементы меню, используемые в инструментальном средстве разработки конструкций геометрической оптики. Исходя из требований пользователя, эти команды представляют действия, которые не связаны с мышью и выполняются как над единым целым. Так как наш проект состоит из объектов, которые объединяются для определенной функции, каждый объект отвечает за то, что он знает о состоянии приложения и внутренних механизмах. Вывод из этого правила заключается в том, что определенные объекты знают, как реагировать на некоторые, но не на все, команды меню. Хороший пример такого поведения приведен из класса `TOpticsApplications`. Экземпляры этого класса контролируют вхождение и прохождение документов, так как объект-приложения находится на более высоком уровне, чем объект-документ. В противоположность этому приложение ничего не знает об атрибутах объектов отображения, потому что объекты отображения находятся на слишком низком уровне абстракции относительно объекта-приложения. Таким образом, объект-приложение является лучшим посредником в системе, способным отвечать за управление только определенными командами меню такими как `New`, `Open` и `Quit`. Большинство стандартных команд (такие, как `Quit`) автоматически управляют `MacApp`. Тем не менее нам необходимо слегка изменить поведение для `New` и `Open`, так как приложение должно управлять не более чем четырьмя документами. Наш проект обеспечивает это в классе `TOpticsDocumentHandler`, поэтому объект управления документом должен работать вместе с объектом-приложением. Следовательно, мы должны заменить

метод DoMenuCommand в классе TOpticsApplications, который мы можем реализовать, следующим образом:

```
function TOpticsApplication.DoMenuCommand (ACmdNumber : CmdNumber) : TCommand;
    override;
begin
    DoMenuCommand := GNoChanges;
    case ACmdNumber of
        cAboutApp:
            DisplayAboutOpticsDialog;
        cNew, cOpen:
            if fOpticsDocumentManager.CanAddDocument then
                DoMenuCommand := inherited DoMenuCommand (ACmdNumber)
            else
                DisplayNoMoreDocumentsDialog;
        kDocumentA .. kDocumentD:
            fOpticsDocumentManager.FocusOnADocument (ACmdNumber);
        cSave, cSaveAs, cSaveCopy:
            fOpticsDocumentManager.RenameDocument
                (TOpticsDocument (GetActiveWindowDocument));
        otherwise
            DoMenuCommand := inherited DoMenuCommand (ACmdNumber);
    end;
end;
```

Реализация данного метода вполне типична для объектно-ориентированных систем: она является краткой и использует методы низкого уровня. Общей формой DoMenuCommand является большой оператор выбора, включающий только те элементы, которыми мы можем локально управлять: константы, такие, как cAboutApp (определенная в MacApp) и kDocumentA (определенная в нашем приложении). Константы kDocumentA через константы kDocumentD представляют собой командные числа, соответствующие первым четырем элементам меню Experiments диалога. Оператор выбора заканчивается конструкцией OtherWise, вызывающей соответствующий метод подкласса для управления командами, которые мы не можем локально обрабатывать.

Отметим, что в данном исходном тексте наиболее интересная работа выполняется посредником, на который указывает поле fOpticsDocumentManager. Объект-приложение и объект-управление документом взаимодействуют очень сложным способом. Например, когда вызывается команда Open и вызов метода CanAddDocument возвращает True, контроль переходит к методу DoMenuCommand суперкласса приложения. В текущем состоянии в MacApp выводится диалог, с помощью которого пользователь может выбрать документ и затем открыть его. Вызов метода AddDocument, определенного для объекта-приложения, осуществляется как часть этого механизма. В нашем проекте интерфейса класса текущий метод AddDocument замещается на собственный, который мы можем реализовать следующим образом:

```
procedure TOpticsApplication.AddDocument (ANewDocument : TDocument); override;
begin
    fOpticsDocumentManager.AddDocument (TOpticsDocument (ANewDocument));
    inherited AddDocument (ANewDocument);
end;
```

Здесь мы в первую очередь ищем цель управления документом, так чтобы соответственно модифицировать Experiments-меню и затем вызвать текущее поведение, определенное в нашем методе суперкласса.

Концептуально объект управления документом должен содержать путь от одного документа к четырем документам. MacApp уже поддерживает несколько открытых документов в глобальной переменной `gDocList`, и поэтому неразумно ссылаться на объект из нескольких других объектов. Тем не менее запомним, что объект управления документом отслеживает свои документы по-другому, чем MacApp: это является фактически проблемно-зависимым поведением. В частности, объекту управления документом нужно знать три параметра для того, чтобы выполнять свою работу: соответствующее командное число для документа, отображенное командное число на определенный элемент меню и число открытых документов.

Данная информация представляет состояние объекта управления документом, и, таким образом, мы можем выбрать представление для данного класса:

```
fCmdToDocMap      : array[kDocumentA .. kDocumentD] of TOptionsDocument;
fCmdToMenuItemMap : array[kDocumentA .. kDocumentD] of Integer;
fNumberOfDocuments : Integer;
```

Первое поле `fCmdToDocMap` представляет отображение командного числа на объект-документ. Это отображение обеспечивает значительно большее содержание информации, чем в простом списке, содержащемся в глобальной переменной `gDocList`. Второе поле `fCmdToMenuItemMap` обеспечивает отображение каждого командного числа на элемент меню. При определении нашего собственного локального состояния мы точно кешируем информацию, которая нам нужна, и заканчиваем эффективной во времени реализацией. Имеется такое же разумное объяснение для включения третьего поля `fNumberOfDocuments`. Мы могли так же легко получить это число путем просмотра списка, содержащегося в `gDocList`. Тем не менее сохранение данного локального состояния делает реализацию более быстрой (мы не вызываем метод другого объекта) и (мы в этом уверены) более понятной. Конечно, компромисс заключается в том, что мы должны сохранять данное локальное состояние согласующимся с соответствующим внешним состоянием.

Отметим, что реализация, которую мы выбрали для класса `TOptionsDocumentManager`, является одним из возможных подходов. Мы выбрали именно это представление, потому что решили пожертвовать простотой для скорости. Если бы после построения приложения мы поняли, что то, что мы сделали, не имеет смысла (возможно, потому, что наше приложение имело ограничения по памяти), то у нас не было иного выхода, как изменить реализацию данного класса. Но поскольку мы спроектировали данный класс, рассматривая его абстрактное поведение до реализации, мы могли выбрать альтернативное представление, не затрагивая ожиданий других объектов-пользователей. Мы будем иметь некоторую рекомпиляцию, но так как мы завершаем изменение только представлением данного класса (который в большей степени невидим), нелогично изменять семантику любого другого объекта или класса, который зависит от `TOptionsDocumentManager`.

В реализации метода `AddDocument` в классе `TOptionsApplications` мы вызываем метод с тем же именем, определенным в классе `TOptionsDocumentManager`. Так как мы выбрали представление, то теперь мы можем реализовать данный метод следующим образом:

```

procedure TOpticsDocumentManager.AddDocument (ADocument : TOpticsDocument);
var
    Index      : Integer;
    Menu       : MenuHandle;
    Title      : Str255;
begin
    for Index := kDocumentA to kDocumentD do
        If (fCmdToDocMap[Index] = nil) then
            begin
                fCmdToDocMap[Index] := ADocument;
                Menu := GetMenu (kExperimentsMenu);
                HLock (Handle (ADocument.fTitle));
                Title := ADocument.fTitle;
                HUnlock (Handle (ADocument.fTitle));
                SetItem (Menu, fCmdToMenuItemMap[Index], Title);
                ADocument.fCommand := Index;
                leave;
            end;
        fNumberOfDocuments := fNumberOfDocuments + 1;
    end;
end;

```

На этом этапе мы модифицируем состояние карт, которые локальны для объекта управления документом, и строим операции низкого уровня (такие, как функция `GetMenu` из Пакета разработчика Macintosh). В результате мы устанавливаем имя соответствующего элемента в `Experiments`-меню при вызове процедуры `SetItem`. Отметим, что мы должны поддерживать отображение документа на его команду для того, чтобы выполнить методы `DeleteDocument` и `RenameDocument`; например, когда документ переименован, мы должны изменить имя элемента меню, соответствующего этому документу путем сохранения командного числа в поле, определенном для этого объекта документа. Как мы узнаем, что нужно включить именно этот элемент меню? Для этого надо воспользоваться методом `DoSetUpMenus` объекта управления документом.

Когда пользователь выбирает элемент меню в `Experiments`-меню, вызов метода `DoMenuCommand` объекта приложения в результате приводит к вызову метода `FocusOnADocument`, определенного в классе `TOpticsDocumentManager`. Данный метод затем использует состояние в таблице командное число/документ (как установлено выше) для активизации документа, соответствующего этому элементу:

```

procedure TOpticsDocumentManager.FocusOnADocument (ACmdNumber : CmdNumber);
begin
    fCmdToDocMap[ACmdNumber].fOpticsWindow.Activate (True);
    fCmdToDocMap[ACmdNumber].fOpticsWindow.Select;
end;

```

Таким образом, объекты-приложения обладают определенной информацией о документах, но только объекты управления документами знают о соответствующих командных числах и документах.

Метод `DoMakeDocument`, определенный в классе `TOpticsApplications`, вызывается при создании или открытии существующего документа. Поэтому в нашей реализации данного метода должен создаваться и инициализироваться объект-документ, который становится частью состояния приложения:

```

function TOpticsApplication.DoMakeDocument
    (fCmdNumber : CmdNumber) : TDocument; override;

```

```

var
  OpticsDocument : TOpticsDocument;
begin
  new (OpticsDocument);
  OpticsDocument.IOpticsDocument;
  DoMakeDocument := OpticsDocument;
end;

```

Здесь опять мы имеем краткую реализацию, построенную на основе абстракций нижнего уровня.

Обращаясь к классу TOpticsDocument, заметим, что он представляет проблемно-зависимую коинкретизацию встроенного в MacApp класса TDocument. Подобно последнему, он участвует в механизме документов MacApp и поэтому должен знать, как считать и записывать оптический эксперимент. Дополнительно объект-документ должен отвечать за управление окнами, связанными с документом. В результате, мы должны поддерживать поля для окон, просматриваемые объекты отображения и палитру, связанную с документом. Как мы уже говорили выше о механизме документов MacApp, мы должны также поддерживать объект, представляющий оптический эксперимент. Таким образом, мы можем завершить реализацию класса, объявив его в определении класса, которое содержится в интерфейсе модуля UOpticsDocuments:

```

fOpticsModel      : TOpticsModel;
fOpticsWindow     : TWindow;
fOpticsView       : TOpticsView;
fPaletteView      : TPaletteView;
fCommand          : Integer;

```

Поле fCommand было получено, так как нам необходимо поддерживать отображение документа на командное число в реализации метода TOpticsDocumentManager.

Когда документ создан или открыт, текущее поведение MacApp сначала создаст объект-документ с помощью метода DoMakeDocument объекта-приложения. Если этот документ не является только что созданным, устойчивое состояние документа считывается с диска, в результате чего вызывается метод DoRead объекта-документа. Обычно мы реализуем данный метод путем построения на основе абстракций нижнего уровня:

```

procedure TOpticsDocument.DoRead (ARefNum : Integer;
                                   RsrcExists,
                                   ForPrinting : Boolean); override;
begin
  inherited DoRead (ARefNum, RsrcExists, ForPrinting);
  fOpticsModel.DoRead (ARefNum);
end;

```

Как для новых, так и для старых документов MacApp следующим привлекает методы DoMakeViews и DoMakeWondows. Опять мы реализуем эти методы путем построения на основе абстракций нижнего уровня:

```

procedure TOpticsDocument.DoMakeViews (ForPrinting : Boolean); override;
var
  PaletteView : TPaletteView;
  OpticsView : TOpticsView;

```

```

begin
  new (PaletteView);
  PaletteView.IPaletteView(self, fOpticsModel);
  fPaletteView := PaletteView;
  new (OpticsView);
  OpticsView.IOpticsView (self, PaletteView, fOpticsModel);
  fOpticsView := OpticsView;
  fOpticsModel.fView := OpticsView;
end;

procedure TOpticsDocument.DoMakeWindows; override;
var
  aWindow : TWindow;
begin
  aWindow := NewPaletteWindow      (kWindowResource, kWantHScrollBar,
                                     kWantVScrollBar, self,
                                     fOpticsView, fPaletteView,
                                     kPaletteImageWidth, kLeftPalette);

  aWindow.AdaptToScreen;
  fOpticsWindow := aWindow;
end;

```

В методе DoMakeWindows мы используем общедоступную процедуру NewPaletteWindow для создания окна согласно проекту, показанному на рис. 9-15.

Как мы видим, в методе DoMakeViews рождение объекта отображения сначала предполагает его создание, а затем его инициализацию. Инициализация объекта в основном существует для установки его первоначального устойчивого состояния. Например, инициализация оптического объекта отображения происходит следующим образом:

```

procedure TOpticsView.IOpticsView (ADocument : TDocument;
                                     APalette : TPaletteView;
                                     AModel : TOpticsModel);
var
  Size : VPoint;
  OpticalBench : TOpticalBench;
  APrintHandler : TOpticsPrintHandler;
begin
  SetVPt(Size, 100, 100);
  IView(ADocument, nil, gZeroVPT, Size, SizeFillPages, SizeFillPages);
  fPaletteView := APalette;
  fOpticsModel := AModel;
  new (OpticalBench);
  OpticalBench.IOpticalBench (self);
  fOpticalBench := OpticalBench;
  fDrawOpticalBench := True;
  fDrawGridLines := False;
  fDrawPageBreaks := False;
  fAutoGridEnabled := True;
  new (APrintHandler);
  APrintHandler.IStdPrintHandler(ADocument, self, False, True, True);
  APrintHandler.fShowBreaks := fDrawPageBreaks;
  APrintHandler.fMinimalMargins := True;
  APrintHandler.fPageDirection := H;
end;

```

Инициализация обычно включает вызов метода инициализации, определенного в суперклассе и следующего перед операторами, которые устанавли-

вают значения различных полей. Мы устанавливаем значения всех полей, чтобы ни один из объектов-пользователей не был захвачен врасплох. Отметим, что мы создаем объект управления выводом на печать как часть инициализации оптического объекта отображения, и он включает знания о том, как выводить на печать содержимое объекта отображения. Так как мы не заботились о текущем изображении на экране страниц прерываний в MacApp, мы создаем свой собственный класс управления выводом на печать так, что мы можем заменять текущее поведение (в частности, мы хотим чтобы номера страниц были изображены во всех углах страницы, а MacApp показывает только некоторые из этих цифр). Почему мы не говорили об этом выше, когда реализовывали интерфейс для большинства модулей? Это объясняется тем, что данный новый класс TOpticsPrintHandler должен быть видимым только для реализации класса оптических объектов отображения. Поэтому мы можем закрыть данный класс от других объектов-пользователей, объявив это в реализации модуля TOpticsViews.

Оптический объект отображения участвует в механизме отображения MacApp и поэтому мы должны заменить текущий метод Draw в интерфейсе нашего класса. Оптический объект отображения должен выводить на экран не только изображение оптического эксперимента, но также различные атрибуты, такие, как метки шкалы и оптическую скамью. Мы должны принять это во внимание при реализации метода Draw:

```
procedure TOpticsView.Draw (Area : Rect); override;
begin
    inherited Draw(Area);
    if (fDrawGridLines and (not gPrinting)) then
        DrawGridLines(Area);
    if fDrawOpticalBench.Draw(Area);
        fOpticalBench.Draw(Area);
    fOpticsModel.Draw(Area, fDrawOpticalBench);
end;
```

Мы могли бы записать исходный текст, который изображает метки шкалы в заданном месте, но путем определения метода DrawGridLines мы упростили реализацию метода Draw и сделали его более понятным. Это является примером совместного действия объектно-ориентированного проектирования и более традиционных методов структурного проектирования. В объектно-ориентированном проектировании мы начинаем с определения классов и объектов, но часто производим декомпозицию сложных методов в небольшие алгоритмические абстракции.

Так же как приложение и объекты-документы, оптический объект отображения участвует в механизме команд меню MacApp. Действительно, оптический объект отображения знает лучше, как реагировать на следующие команды: Cut, Copy, Show Optical Bench и Show Page Breaks. MacApp делает вызов метода DoSetUpMenus для каждого объекта, включенного в командную цепочку. Для этого необходимо сделать доступными команды, которые для него представляют интерес. Итак, мы можем реализовать данный метод для класса TOpticsView следующим образом:

```
procedure TOpticsView.DoSetUpMenus; override;
var Count : Integer;
begin
    inherited DoSetUpMenus;
```



```

Count := fOpticsModel.NumberOfSelections;
Enable(cCut, (Count > 0));
Enable(cCopy, (Count > 0));
CanPaste(kClipType);
Enable(cClear, (Count > 0));
Enable(cCopy, (Count > 0));
CanPaste(kClipType);
Enable(cClear, (Count > 0));
Enable(cSelectAll, (Count < fOpticsModel.NumberOfLenses));
Enable(kDrawingSize, True);
Enable(kShowOpticalBench, True);
SetMenuItemState (kShowOpticalBench, kStringResource, kDrawOpticalBench,
                  kDontDrawOpticalBench, fDrawOpticalBench);
Enable(kShowGridLines, True);
SetMenuItemState (kShowGridLines, kStringResource, kDrawGridLines,
                  kDontDrawGridLines, fDrawGridLines);
Enable(kShowPageBreaks, True);
SetMenuItemState (kShowPageBreaks, kStringResource, kDrawPageBreaks,
                  kDontDrawPageBreaks, fDrawPageBreaks);
Enable(kEnableAutoGrid, True);
SetMenuItemState (kEnableAutoGrid, kStringResource, kUseAutoGrid,
                  kDontUseAutoGrid, fAutoGridEnabled);
Enable(kLensSpecification, (Count = 1));

```

end;

Заметим, что нам предоставлены только те элементы меню, на которые оптические объекты отображения могут реагировать, и критерий, по которому нам предоставляется команда, различен для каждого элемента меню. Например, предоставляется команда Cut, только если существует более чем один выбранный объект в оптической модели. С другой стороны, предоставляется команда Select All, только если в модели существуют некоторые объекты, которые еще не выбраны. Такие элементы меню, как Show Optical Bench, предоставляются согласно значению соответствующего поля в оптических объектах отображения. Вызов процедуры SetMenuItemState считывает одно из двух имеющихся элементов меню из ресурсного файла согласно булевой величине, которая дается как параметр.

Если объекту предоставлен индивидуальный элемент меню, он в общем случае будет также обрабатывать эту команду всякий раз, когда выбран данный элемент меню. Это правило, в частности, справедливо для класса TOpticsView. В классе TOpticsApplication мы выполняем метод DoMenuCommand, вызывая абстракции нижнего уровня. Мы будем выполнять то же самое и здесь, но дважды, так как команды, которые могут обработать оптический объект отображения, являются тривиальными. Например, при выборе команды ShowOpticalBench (или Hide OpticalBench в зависимости от состояния поля fDrawOpticalBench) просто переключается состояние поля и изменяется соответствующий атрибутивный объект отображения. Нет необходимости в том, чтобы эта команда была отменяемой, так как пользователь может сделать обратное действие, выбрав снова тот же элемент меню. Следовательно, создавать объект-команду для выполнения этих действий излишне. С другой стороны, все команды редактирования, включая Cut, Show, Paste и Clear, являются отменяемыми. Пользователю, который «вырежет» линзу и затем поймет, что это действие было неправильным, будет предоставлена возможность отменить команду и, таким образом, вернуть эксперимент в первоначальное состояние. Так как объекты-команды содержат знания о том, как выполнять, отменять и переделывать действие, хорошо,

что наше приложение создает объекты-команды, которые действуют как посредники, отвечающие за выполнение этих действий.

Реализация метода DoMenuCommand класса TOpticsView является долгой процедурой, потому что существует слишком много команд, которыми он может управлять. Концептуально реальная работа производится где-нибудь в другом месте:

```
function TOpticsView.DoMenuCommand (ACmdNumber : CmdNumber) : TCommand; override;
```

```
var
```

```
  Proceed           : Boolean;
  CurrentModelSize  : Point;
  CurrentDirection  : VHSelect;
  CurrentResolution : Point;
  CurrentPageSize   : VPoint;
  CurrentViewSize   : VPoint;
  AcceptChanges     : Boolean;
  PageStrips        : Point;
  SelectedLens      : TLens;
  NewFocallLength   : FocallLength;
  CutCommand        : TCutCommand;
  CopyCommand       : TCopyCommand;
  PasteCommand      : TPasteCommand;
  ClearCommand      : TClearCommand;
```

```
function CheckLens (Item : TLens) : Boolean;
```

```
begin
```

```
  CheckLens := Item.IsSelected;
```

```
end;
```

```
begin
```

```
  DoMenuCommand := gNoChanges;
```

```
  case ACmdNumber of
```

```
    cSelectAll:
```

```
      fOpticsModel.SelectAll;
```

```
    cCut:
```

```
      begin
```

```
        new(CutCommand);
```

```
        CutCommand.ILensCommand(self, fOpticsModel, False);
```

```
        DoMenuCommand := CutCommand;
```

```
      end;
```

```
    cCopy:
```

```
      begin
```

```
        new(CopyCommand);
```

```
        CopyCommand.ILensCommand(self, fOpticsModel, False);
```

```
        DoMenuCommand := CopyCommand;
```

```
      end;
```

```
    cPaste:
```

```
      begin
```

```
        new(PasteCommand);
```

```
        PasteCommand.ILensCommand(self, fOpticsModel, False);
```

```
        DoMenuCommand := PasteCommand;
```

```
      end;
```

```
    cClear:
```

```
      begin
```

```
        new(ClearCommand);
```

```
        ClearCommand.ILensCommand(self, fOpticsModel, False);
```

```
        DoMenuCommand := ClearCommand;
```

```
      end;
```

```
    kDrawingSize:
```

```

begin
    CurrentModelSize := fOpticsModel.CurrentSize;
    CurrentDirection :=
        TStdPrintHandler(fPrintHandler).fPageDirection;
    CurrentResolution := fPrintHandler.fEffectiveDeviceRes;
    CurrentPageSize := fPrintHandler.fViewPerPage;
    CurrentViewSize := fSize;
    DisplayDrawingSizeDialog (CurrentModelSize, CurrentResolution,
                              CurrentPageSize, CurrentViewSize,
                              CurrentDirection, AcceptChanges);

    if AcceptChanges then
        begin
            if not EqualVPt(CurrentViewSize, fSize) then
                Resize(CurrentViewSize.H, CurrentViewSize.V, True);
            if (CurrentDirection <>
                TStdPrintHandler(fPrintHandler).fPageDirection) then
                TStdPrintHandler(fPrintHandler).fPageDirection :=
                    CurrentDirection;
            ForceRedraw;
        end;
    end;
kShowOpticalBench:
    ToggleDrawOpticalBench;
kShowGridLines:
    ToggleDrawGridLines;
kShowPageBreaks:
    ToggleDrawPageBreaks;
kEnableAutoGrid:
    ToggleAutoGridEnabled;
kLensSpecification:
    begin
        if (fOpticsModel.NumberOfSelections = 1) then
            begin
                SelectedLens :=
                    TLens(fOpticsModel.FirstLensThat (CheckLens));
                DisplayLensSpecificationDialog (SelectedLens,
                                                NewFocalLength,
                                                AcceptChanger);

                if AcceptChanges then
                    begin
                        SelectedLens.SetFocalLength (NewFocalLength);
                        fDocument.fChangeCount := fDocument.fChangeCount + 1;
                    end;
            end;
        otherwise
            DoMenuCommand := Inherited DoMenuCommand (ACmdNumber);
    end;
end;
end;

```

В данном методе мы управляем командами Select All, Drawing Size..., Show Optical Bench, Show Grid Lines, Show Page Breaks, Enable Autogrid и Lens Specification... в соответствующих местах. Эти команды не являются отменяемыми, но все другие команды — отменяемы. Таким образом, выполнение различных команд редактирования сначала включает создание объекта-команды соответствующего класса и затем его инициализацию. Только что созданный объект-команда возвращается обратно как результат функции метода DoMenuCommand. Согласно механизму команд меню MacApp, метод PerformCommand объекта-приложения в конечном счете получает данный

объект-команду, а затем метод PerformCommand вызывает DoIt объекта-команды.

Заметим, что наш метод содержит некоторое количество исходного текста для выполнения команд Drawing Size... и Lens Specification.... В обоих случаях выполнение включает нахождение определенного состояния, необходимого для диалога, вызов общедоступной процедуры, соответствующей этому диалогу, затем изменение состояния объекта отображения, если пользователь в действительности редактировал диалог и устанавливал его новые значения. Эти преимущества использования ресурсов диалогов очевидны.

Метод DoMouseCommand участвует в механизме действия мыши MacApp и с точки зрения его реализации он похож на метод DoMenuCommand. Поскольку мы хотим, чтобы все действия мыши — выбор, перемещение, выбор картинки сервисной программы — были отменяемыми, данный метод должен создавать один из нескольких объектов-команд. Как он узнает, какой объект-команду создавать? Ответ на этот вопрос зависит от состояния объекта-палитры. Если пользователь выбрал для линзы определенную сервисную программу, действие мыши в оптическом объекте отображения представляет начало работы с эскизом линзы. Если пользователь выбрал сервисную программу, позволяющую указывать на линзы, действие мыши в оптическом объекте отображения представляет собой групповой выбор (если курсор мыши не находится под какой-либо линзой) или начало перемещения (если существуют выбранные линзы или курсор мыши находится под линзой). Мы можем выразить данный алгоритм следующим образом:

```
function TObjectView.DoMouseCommand (var TheMouse : Point;
var Info : EventInfo;
var Hysteresis : Point) : TCommand; override;

var
  SketcherCommand      : TSketcherCommand;
  SelectorCommand      : TSelectorCommand;
  DraggerCommand       : TDraggerCommand;
  LensUnderMouse       : TLens;
procedure CheckLens (Item : TLens);
begin
  if PInRect (TheMouse, Item.Frame) then
    LensUnderMouse := TItem;
end;
begin
  DoMouseCommand := gNoChanges;
  if fPaletteView.IsPointer then
    begin
      LensUnderMouse := nil;
      fOpticsModel.EachLens(CheckLens);
      if LensUnderMouse = nil then
        begin
          fOpticsModel.DeselectAll;
          new (SelectorCommand);
          SelectorCommand>ILensCommand (self, fOpticsModel,
            fDrawOpticalBench);
          DoMouseCommand := SelectorCommand;
        end
      else
        begin
          if not Info.TheShiftKey then
            fOpticsModel.DeselectAll;
          if (Info.TheShiftKey and LensUnderMouse.IsSelected) then
```

```

        LensUnderMouse.Deselect
    else
        LensUnderMouse.Select;
    if (fOpticsModel.NumberOfSelections > 0) then
        begin
            new (DraggerCommand);
            DraggerCommand.ILensCommand := (self, fOpticsModel,
                fDrawOpticsBench);
            DraggerCommand.fConstrainsMouse :=
                fAutoGridEnabled;
            DoMouseCommand := DraggerCommand;
        end;
    end
else
    begin
        fOpticsModel.DeselectAll;
        new (SketcherCommand);
        SketcherCommand.ILensCommand (self, fOpticsModel, fDrawOpticalBench);
        SketcherCommand.fLens := fPaletteView.CurrentTool;
        SketcherCommand.fLens.fOpticsModel := fOpticsModel;
        SketcherCommand.fConstrainsMouse := fAutoGridEnabled;
        DoMouseCommand := SketcherCommand;
    end;
end;
end;

```

Рассмотрим, как оптический объект отображения объединяется со своим объектом-палитрой. Объект-палитра знает только, как управлять выбором сервисной программы, но он не знает, что делать с таким выбором. Объект-палитра передает оптическому объекту отображения информацию о том, какая сервисная программа выбрана.

Как изображается объект-палитра? Можно использовать множество подходов, и самый простой из них — определить картинку, которую изображает объект-палитра. Для того чтобы создать картинку, можно использовать любую программу, основанную на закрашивании пикселей (например, MacPaint), и затем вставить это изображение в ресурсный файл приложения, используя такую сервисную программу, как ResEdit. Это опять тот случай, когда мы отделяем внешне наблюдаемые характеристики от исходного текста. В общем случае мы должны редактировать картинку много раз, пока ее вид не удовлетворит нашему представлению, но у нас имеется возможность сделать это без какой-либо перекомпиляции исходного текста и в конечном счете без воздействия на проект программы.

Данное проектное решение приводит к следующему представлению класса:

```

fCurrentTool      : 0 .. kKindsOfLenses;
fImage            : PicHandle;
fImageRect        : Rect;
fViewRect         : Rect;

```

Последние три поля включают состояние картинки, как оно считается из ресурсного файла в ходе инициализации объекта-палитры. Таким образом, изображение палитры становится тривиальным:

```

procedure TPaletteView.Draw (Area : Rect);
begin

```

```

PenNormal;
DrawPicture (fImage, fImageRect);
MoveTo ((kPaletteImageWidth - 1), 0);
LineTo ((kPaletteImageWidth - 1), kPaletteImageWidth);
end;

```

При нескольких вызовах к программам QuickDraw, мы сначала устанавливаем характеристики карандаша, выводим изображение картинки и затем рисуем линию, отделяющую палитру от просматриваемого объекта отображения. Наблюдательный читатель увидит, что мы использовали константу kPaletteImageWidth при реализации метода DoMakeWindows, определенного для класса TOpticsDocument. Действительно, это классический пример того, почему мы объявляем константы вместо использования числовых литеров в нашей программе: константы помогают сделать нашу программу более понятной (а наши проектные решения более точными), а также более удобной для поддержки, увеличивая долю общих статических величин среди абстракций.

При рассмотрении проекта данного класса интересной является функция CurrentTool. В нашем проекте CurrentTool возвращает объекту-лине соответствующую сервисную программу из палитры, выбранной в настоящий момент. Оптический объект отображения затем использует данный объект в методе DoMouseDown для инициализации выбора картинка сервисной программы объекта-команды. В нашей реализации CurrentTool можно использовать большой оператор выбора, оцениваемый для выбранного в настоящий момент элемента, и затем создавать соответствующую линию. Но выполнение некоторого определенного для класса действия при простом рассмотрении объекта класса приводит к некоторым противоречиям с объектно-ориентированным стилем: зачем проверять класс объекта, когда сам объект содержит эти знания? Вместо этого при нашем подходе создаются прототипы для каждой возможной линии, когда начинается работа с приложением, а затем просто делаются копии линз, которые нам понадобятся. По этой причине мы объявляем массив PrototypeLenses в интерфейсе модуля UOpticsModels, который мы инициализируем таким образом, чтобы каждый элемент массива указывал на объект другого класса линз. Таким образом, данный массив формирует таблицу отображения сервисной программы на линзу. Его лучше всего определить в модуле UOpticsModels, но так как он должен быть видимым другим модулям, мы должны включить следующие объявления в интерфейс модуля:

```

const
    kKindsOfLenses = 6;
...
var
    PrototypeLenses : array[1 .. kKindsOfLenses] of TLens;

procedure InitializeOpticsModels;

```

Общедоступная процедура InitializeOpticsModels аналогична процедуре InitializeOpticsDialogs, определенной в модуле UOpticsDialogs. InitializeOpticsModels вызывается из основной программы для создания и инициализации прототипа экземпляра каждой линзы, как показано в следующем фрагменте:

```

var
    DoubleConvexLens : TDoubleConvexLens;
begin
    new (DoubleConvexLens);
    DoubleConvexLens.IShape (nil);
    PrototypeLenses [kDoubleConvexLensID] := DoubleConvexLens;
end;

```

Отметим, что мы опять предпочитаем использование таких констант, как `kDoubleConvexLensID` численным литерам. Выполняя реализацию метода `CurrentTool`, мы должны записать:

```

function TPaletteView.CurrentTool : TLens;
begin
    CurrentTool := TLens (PrototypeLenses [fCurrentTool].Clone);
end;

```

Для того чтобы закончить с реализацией интерфейса пользователя, мы включаем диалоги, определенные в модуле `UOpticsDialogs` (рис. 9-16). Реализация общедоступной процедуры `DisplayDrawingSizeDialog` приводит нас к введению некоторых проблемно-зависимых классов диалогов, которые включают желаемое поведение. Побочным действием данного рекурсивного проектирования является то, что мы заканчиваем проектирование, имея некоторые мощные классы, которые мы могли бы использовать для других приложений. Другие диалоги, такие, как `Lens Specification...` (рис. 9-16), будут иными проще. Например, последний приводимый диалог сначала включает создание из шаблона соответствующего окна, затем установление начальных значений для каждого управления диалогом и в конце передачу управления пользователю (метод `PoseModally`). Когда пользователь убирает диалог, управление возвращается к методу, и формальные параметры функции устанавливаются следующим образом:

```

procedure DisplayLensSpecificationDialog (SelectedLens : TLens;
var NewFocalLength : FocalLength;
var AcceptChanges : Boolean);
var
    Window : TWindow;
    DialogView : TLensSpecificationDialogView;
    Dismissal : IDType;
    AString : AString;
begin
    Window := NewTemplateWindow (kLensSpecificationResource, nil);
    DialogView := TLensSpecificationDialogView (Window.FindSubView ('DLOG'));
    DialogView.fPicture := SelectedLens.fPicture;
    NumToString ((SelectedLens.fCenter.H - kOpticalBenchHOffset), AString);
    DialogView.ParamTxt ('^0', AString);
    GetIndString (AString, kStringResource, SelectedLens.fName);
    DialogView.ParamTxt ('^1', AString);
    if SelectedLens.fConverging then
        begin
            TNumberText (DialogView.FindSubView ('foci')).fMinimum := 0;
            TNumberText (DialogView.FindSubView ('foci')).fMaximum := MaxInt;
        end
    else
        begin
            TNumberText (DialogView.FindSubView ('foci')).fMinimum := MaxInt;

```

```

TNumberText (DialogView.FindSubView ('focl')).fMaximum := 0;
end;
TNumberText (DialogView.FindSubView ('focl')).
  SetValue (SelectedLens.fFocalLength, kDontRedraw);
DialogView.SelectEdit (Text ('focl', True);
Dismitter := DialogView.PoseModally;
if (Dismitter = 'ok ') then
  NewFocalLength :=
    FocalLength (TNumberText (DialogView.FindSubView ('focl')).GetValue);
AcceptChanges := (Dismitter = 'ok ') and
  (NewFocalLength <> SelectedLens.fFocalLength);
Window.Close;
end;

```

Для того чтобы иметь законченную, выполняемую программу, мы должны создать основную программу. Создание основной программы как последнего действия по реализации кажется невероятно странным для тех, кто придерживается методов структурного проектирования. Однако в объектно-ориентированных системах корень системы для нас не особенно важен; нас интересует прежде всего, как найти место для объектов самого высокого уровня. Итак, для инструментального средства разработки конструкций геометрической оптики мы можем написать основную программу MOptics следующим образом:

```

programm Optics;
uses
  UMacApp, UPrinting, UDialog, UTeVew,
  UOpticsModels, UOpticsDialog, UOpticsControllers, UOpticsViews,
  UOpticsDocument, UOptics;
{$S Main}
var
  gOpticsApplication : TOpticsApplication;
begin
  InitUMacApp (10);
  InitPrinting;
  InitializeOpticsModels;
  InitializeOpticsDialogs;
  InitializeOpticsViews;
  new (gOpticsApplication);
  gOpticsApplication.IOpticsApplication;
  gOpticsApplication.Run;
end;

```

Теперь мы можем создать исполняемый прототип для отработки наиболее важных элементов интерфейса пользователя. Мы полностью поддерживаем данную раннюю интеграцию, для того чтобы мы могли обнаружить любой серьезный недостаток в интерпретации требований или в выборе механизмов.

Реализация модели

Если нас удовлетворяет интерфейс пользователя, мы можем завершить проектирование и реализацию классов, содержащихся в модулях UOpticsControllers и UOpticsModels. Действительно, если мы имеем достаточно ресурсов разработки, можно развивать интерфейс пользователя и его глубинную модель параллельно.

Реализация подклассов класса TShape. Сначала выберем представление для класса TLens. Мы используем подход, подобный тому, который мы применяли для класса TPaletteView. Таким образом, мы будем создавать картинку для каждой линзы (поэтому их можно легко изменять), и метод Draw будет реализовывать изображение этой картинки. Кроме полей, требующихся для содержания ресурсов данной картинки, нам необходимо дополнительное состояние. В частности, мы должны поддерживать простую идентификацию класса линз для того, чтобы использовать ее, когда происходит считывание и запись состояния документа. Так как Object Pascal не поддерживает устойчивость объекта, мы должны свертывать его путем кодирования класса каждого объекта в файл данных документа. Необходимо также помнить текущее и последнее положение линзы, и выбрана ли она в настоящий момент. Наконец, мы должны помнить фокусное расстояние линзы, является ли она собирающей или рассеивающей (что влияет на допустимый диапазон значений ее фокусного расстояния), и ее научное название (так, чтобы мы могли изобразить ее в Lens Specification... диалоге). Получив эти параметры, мы можем завершить реализацию интерфейса класса TLens следующим образом:

```
fID      : Integer;
fCenter  : Point;
fDraggingCenter : Point;
fEltRect : Rect;
fIsSelected : Boolean;
fPicture : PicHandle;
fFocalLength : Integer;
fConverging : Boolean;
fName    : Integer;
```

Так как TLens представляет в нашей проблеме наиболее низкий уровень абстракции, его реализация завершается выполнением значительного объема работы. Например, считыванием и записью документа управляет объект-приложение, но знаем о том, как считать и записать индивидуальный объект-линзу, обладает только сам объект-линза. Таким образом, мы можем реализовать методы DoRead и DoWrite объекта линзы, которые должны дополнять друг друга:

```
procedure TLens.DoRead (ARefNum : Integer);
var
    Size      : LongInt;
    TheCenter  : Point;
    TheFocalLength : FocalLength;
begin
    Size := SizeOf (Point);
    FailOSErr (FSRead (ARefNum, Size, @TheCenter));
    fCenter := TheCenter;
    Size := SizeOf (FocalLength);
    FailOSErr (FSRead (ARefNum, Size, @TheFocalLength));
    fFocalLength := TheFocalLength;
end;

procedure TLens.DoWrite (ARefNum : Integer);
var
    TheLensData : LensData;
    Size        : LongInt;
```

```

begin
    TheLensData.TheId := fId;
    TheLensData.TheCenter := fCenter;
    TheLensData.TheFocalLength := fFocalLength;
    Size := SizeOf (LensData);
    FailOnError (FSWrite (ARefNum, Size, @TheLensData));
end;

```

Данные методы не вызываются непосредственно объектом-приложением; они вызываются объектом оптическая модель, которому принадлежат линзы. Например, метод DoRead класса TOpticsModel представлен следующим образом:

```

procedure TOpticsModel.DoRead (ARefNum : Integer);
var
    Size      : LongInt;
    Count     : Integer;
    TheID     : Integer;
    ALens     : TLens;
    Index     : Integer;
begin
    Size := SizeOf (Integer);
    FailOnError (FSRead (ARefNum, Size, @Count));
    for Index := 1 to Count do
        begin
            Size := SizeOf (Integer);
            FailOnError (FSRead (ARefNum, Size, @TheID));
            ALens := TLens (PrototypeLenses [TheID].Close);
            ALens.fOpticsModel := self;
            ALens.DoRead (ARefNum);
            fLenses.InsertLens
        end;
        TraceRays;
    end;
end;

```

Обратите внимание, что данный метод использует тот же массив PrototypeLenses, который мы использовали в классе TPaletteView. После того как модель была считана с диска, самым последним действием является процесс вычерчивания луча; короче говоря, мы приходим обратно к данному методу.

Реализация команд. Так как мы завершили реализацию методов различных объектов-линз и объектов оптических моделей, которые нужны для механизма отображения MacApp, мы можем завершить реализацию классов команд. Например, рассмотрим класс TSketcherCommand, который отвечает за размещение новой линзы на оптической скамье и за перемещение мыши, пока пользователь передвигает линзу в ее положение. Данный класс является подклассом TLensCommand и добавляет одно поле fLens к состоянию его объектов. Объект-команда «картинка» сервисной программы создается при вызове метода DoMouseDownCommand, определенного в классе TOpticsView. Инициализация объекта-команды «картинка» сервисной программы является простой: сначала мы вызываем метод его суперкласса (который инициализирует поля, определенные в его суперклассе), следующим инициализируется объект-команда с текущим состоянием MacApp, и в конце инициализируется локальное поле fLens. Мы можем выразить это следующим образом:

```

procedure TSketcherCommand.ILensCommand (ASuperView : TView;
    AnOpticsModel : TOpticsModel;

```

```

                                DrawPosition : Boolean); override;
begin
    inherited ILensCommand (ASuperView, AnOpticsModel, DrawPosition);
    ICommand (kSketcherCommand, ASuperView.fSuperView.fDocument,
              ASuperView, TScroller (ASuperView.fSuperView));
    fLens := nil;
end;

```

В классе TCommand существует поле fChanges, которое инициализируется True командой ICommand. Когда объект-приложение выполняет команду, он увеличивает счетчик изменений соответствующего объекта-документа (поле fChangeCount), только если поле fChanges содержит True. Вспомним из предыдущей дискуссии по поводу механизма документов MacApp, что MacApp проверяет поле fChangeCount объекта-документа, прежде чем закрыть его, для того чтобы дать пользователю возможность сохранить любые изменения. Это будет именно то, что нам следует ожидать. Такие действия, как выбор картинки сервисной программы, вырезание, вставка, перемещение изменяют состояние объекта-документа, и мы не хотим, чтобы пользователь по невнимательности потерял такие изменения. Здесь мы имеем еще один пример, как система может показывать очень сложное поведение с помощью некоторых очень простых механизмов.

Как мы видим, в реализации метода DoMouseCommand, определенного в классе TOpticsView, после того как оптический объект отображения создаст объект-команду «картинку» сервисной программы, объект отображения инициализирует поле fLens картинки сервисной программы объектом, из который указывает текущая сервисная программа палитры. В нашей реализации метода CurrentTool fLens, таким образом, указывает на имитацию прототипа линзы.

Метод DoIt выполняется объектами класса TSketcherCommand там, где получается реальное действие. По существу мы должны вставить новую линзу в оптическую модель, поместить ее как выбранную и затем получить новый путь луча. Мы можем выразить данный алгоритм следующим образом:

```

procedure TSketcherCommand.DoIt; override;
begin
    fOpticsModel.AddLens (fLens);
    fLens.Select;
    fOpticsModel.TraceRays;
end;

```

UndoIt и RedoIt вызываются объектом-приложением согласно управлению MacApp командой Undo в Edit-меню. Действие UndoIt заключается в простом обращении действия DoIt, а действие RedoIt заключается в обращении действия UndoIt. Таким образом, мы можем написать:

```

procedure TSketcherCommand.UndoIt; override;
begin
    if fView.Focus then
        begin
            fOpticsModel.DeselectAll;
            fOpticsModel.RemoveLens (fLens);
            fOpticsModel.TraceRays;
        end;
end;
procedure TSketcherCommand.RedoIt; override;

```

```

begin
    if fView.Focus then
        begin
            fOpticsModel.DeselectAll;
            DoIt;
        end;
    end;
end;

```

MacApp определяет состоящий из двух частей процесс для выполнения команды. До того как метод `HandleEvent` объекта-приложения выполняет новую команду (через метод `DoIt`), `HandleEvent` сначала передаст последнюю команду (через вызов метода `Commit`). Объект-команда может быть создан и затем отменен, и поэтому он никогда не передается. В любом случае перед попыткой выполнить новую команду, MacApp освобождает последнюю команду так, что старые объекты-команды не накапливаются во времени. Таким же способом объект-команда может использовать медленную эволюцию; это означает, что он сможет передать методу `Commit` часть своей работы, которую будет сложно или невозможно отменить или выполнить снова, например такую, как изменение состояния очень большого документа. Когда команда-картинка сервисной программы выполнена (через метод `DoIt`), объект, на который указывает поле `fLens`, представляет собой два совместно используемых объекта: сам объект-команда «картинка» сервисной программы и объект оптическая модель. Если команда-картинка сервисной программы отменена, то, когда мы освобождаем объект-команду, мы также должны освободить линзу, на которую указывает поле `fLens`. Таким образом, мы должны написать:

```

procedure TSketcherCommand.Free; override;
begin
    FreeObject (fLens);
    Inherited Free;
end;

```

`FreeObject` является подпрограммой, определенной в MacApp. В этой подпрограмме сначала проверяется, равно ли нулю значение числа ссылок на данный объект, а если нет, вызвать его метод `Free`.

Когда команда-картинка сервисной программы передана, объект-команда должен прервать его связь с объектом-линзой, так чтобы, при освобождении команды, мы не разрушили линзу. Поэтому был использован предшествующий вызов подпрограммы `FreeObject`. Таким образом, метод `Commit` можно завершить следующим образом:

```

procedure TSketcherCommand.Commit; override;
begin
    fLens := nil;
end;

```

Объект-команда «картинка» сервисной программы должна выполнять еще две важные функции: она должна соответствующим образом перемещать мышь, когда пользователь перемещает линзу в определенное место, а также обеспечивать некоторые визуальные подсказки, чтобы пользователь мог сказать, где именно объект располагается на оптической скамье. Как часть процесса обработки, включенная в управление нажатием клавиши мыши, объект-приложение предоставляет текущей команде возможность перемещать

мышь, вызывая его собственный метод `TrackMouse`. При перемещении мыши объект-приложение повторно вызывает метод `TrackFeedback` объекта, который управляет перемещением мыши (он обычно является объектом-командой). Таким образом, эти два метода должны работать совместно, чтобы создать у пользователя иллюзию, что он сам перемещает линзу по экрану.

Действие мыши состоит из трех частей: нажатие клавиши мыши, перемещение мыши, отжатие клавиши мыши. Большинство команд могут игнорировать отжатие клавиши мыши (исключение составляют некоторые команды, которые должны рисовать объекты, состоящие из множества частей, например, многоугольники). Когда обнаружено нажатие клавиши мыши, метод `TrackMouse` заставляет обновляться объект отображения и новая линза появляется впервые. Таким образом, при движении мыши метод `TrackMouse` восстанавливает область под мышью. Затем метод `TrackFeedback` изображает линзу снова, но в другом месте. Концептуально можно представить метод `TrackMouse` как процесс, который убирает линзу, а метод `TrackFeedback` как процесс, который изображает новую линзу. Так как `MacApp` повторяет данный процесс как часть механизма главного цикла событий, то пользователю кажется, что объект начинает медленно перемещаться по экрану. Мерцание обычно обнаруживается, когда рисование объекта занимает много времени. Это объясняет существование метода `TrackFeedback`. Обычно, когда перемещается объект, пользователю не нужно видеть объект целиком, а только каркас основных его частей. В нашем приложении мы можем целиком рисовать линзу, потому что вывод единственной картинке на экран не требует значительных ресурсов компьютера.

Теперь мы можем завершить реализацию двух методов класса `TSketcherCommand`:

```

procedure TSketcherCommand.TraceFeedback (AnchorPoint,
                                           NextPoint : VPoint;
                                           TurnItOn,
                                           MouseDidMove : Boolean); override;

var
  APoint : Point;
begin
  if MouseDidMove then
    begin
      PenMode (PatOR);
      APoint := VPToPt (NextPoint);
      APoint.V := kOpticsAxis;
      APoint.H := Max (APoint.H,
                      (kOpticalBenchHOffset + kMinimumHOffset));
      APoint.H := Min (APoint.H,
                      (fView.fSize.H — kOpticalBenchHOffset
                       — kMinimumHOffset));

      fLens.fCenter := APoint;
      fLens.DrawLens (fDrawPosition);
    end;
end;

function TSketcherCommand.TraceMouse (ATrackPhase : TrackPhase;
                                       var AnchorPoint,
                                       PreviousPoint,
                                       NextPoint : VPoint;
                                       MouseDidMove : Boolean); TCommand; override;

```

```

begin
  TrackMouse := self;
  if (ATrackPhase = TrackPhase) then
    begin
      fView.GetWindow.Update;
      if fView.Focus then;
    end;
  else if ((ATrackPhase = TrackMove) and MouseDidMove) then
    begin
      fLens.Invalidate;
      fView.GetWindow.Update;
      if fView.Focus then;
    end;
end;
end;

```

В теле метода `TrackFeedback` мы должны убедиться, что линза не перемещается за пределы оптической скамьи и всегда располагается вдоль оптической оси (горизонтальной линии, обозначаемой константой `kOpticalAxis`).

Так как требования предусматривают, чтобы пользователь мог ограничивать перемещение мыши метками шкалы, расстояние между которыми равняется пяти точкам (через `Enable Autogrid` команду меню), все команды должны выполнять метод `TrackConstrain`. Таким образом, предполагая, что можно ограничивать перемещения мыши, до того как вызываются методы, как `TrackMouse` и `TrackFeedback`, `MacApp` сначала вызывает метод `TrackConstrain`, чтобы позволить объекту ограничить перемещение мыши определенной областью или определенным дискретом перемещения. Все команды должны совместно использовать данное поведение, поэтому мы ввели класс `TLensCommand`, в котором реализация метода `TrackConstrain` используется всеми объектами-командами:

```

procedure TLensCommand.TrackConstrain
  (AnchorPoint,
   PreviousPoint : VPoint;
   var NextPoint : VPoint); override;

begin
  NextPoint.H := (NextPoint.H div kAutoGridSize) * kAutoGridSize;
  NextPoint.V := (NextPoint.V div kAutoGridSize) * kAutoGridSize;
end;

```

Механизм буфера `MacApp` связан с обработкой команд. В стандарте интерфейса пользователя `Macintosh` предполагается, что все приложения по возможности поддерживают общие команды редактирования: вырезание, копирование, чистку и вставку. Требования в нашей задаче также определяют данное поведение, и это объясняет, почему мы имеем специализированные классы `TCutCommand`, `TCopyCommand`, `TClearCommand`, `TPasteCommand`.

Когда создается объект-команда «вырезать» (в методе `DoMenuCommand` оптического объекта отображения), мы инициализируем его, используя все линзы, выбранные в настоящий момент в объекте отображения. Когда мы выполняем данную команду, в результате ее действия линзы убираются из оптической модели, появляется новый путь луча, и затем все линзы сохраняются так, что они могут быть вставлены в этот или другой объект отображения. Команда «копирование» выполняет подобное действие помимо того, что она копирует линзы, но она не убирает их из модели. Команда «очистить» только убирает выбранные линзы из модели; она не копирует их. Когда создается команда «вставить» (которая возможна только тогда,

когда имеется что вставить), она «заглядывает» в то же место, где последняя команда «копировать» или «вырезать» сохранила свое состояние.

Общее хранилище вырезанных и скопированных объектов называется буфером, и каждое приложение обычно имеет только один буфер. В основном буфер подобен любому объекту отображения, хотя пользователь не может его редактировать, но может его просматривать (с помощью ShowBuffer). Мы должны поддерживать список объектов, которые были вырезаны или скопированы и, следовательно, могут быть вставлены. Это означает, что буфер должен иметь возможность обрабатывать простые объекты, такие, как текст, или более проблемно-зависимые, такие, как линзы. Буфер также позволяет копировать или вставлять объекты в другие приложения. Например, мы можем вырезать линзы из инструментального средства разработки конструкций геометрической оптики и вставить его в приложение для рисования; фактически так создано большинство рисунков в данной главе.

Так как буфер должен обрабатывать объекты проблемно-зависимых классов, мы должны включить удобный класс TClipboardView в наше приложение. Но поскольку он аналогичен классу TOpticsView, мы не будем рассматривать реализацию и использование.

Применение алгоритмической декомпозиции. Предполагая, что мы закончили реализацию команд всех классов, мы остановимся на одном методе, с которого все проектировщики должны будут начинать: методе TraceRays, определенном в классе TOpticModel. Почему мы выбрали данный метод? По-видимому, он потребует большого объема математических вычислений, необходимых для данного приложения. Это действительно так, но наши концептуальные принципы для всех объектно-ориентированных систем состоят в том, что мы рассматриваем мир как объединение объектов, которые соединяются друг с другом для достижения желаемой функциональности. Управление, основной центр структурного проектирования, не является наиболее важной для нас целью.

На данном этапе, когда мы имеем все ключевые абстракции, мы можем воспользоваться методом TraceRays. Если мы предполагали, что данный алгоритм имеет высокий процент риска, мы должны были параллельно с нашими разработками привлечь к работе физиков для прототипирования алгоритма. Но оказалось, что алгоритм данного метода достаточно прост; при условии что известны основные механизмы, которые мы уже ввели.

По законам геометрической оптики луч начинается от реального объекта, затем проходит от линзы к линзе в порядке увеличения их расстояния от действительного объекта. Это объясняет, почему мы сохраняли состояния линз в оптической модели, используя список fLenses, который является полем класса TLensList. Класс TLensList является подклассом класса TSortedList, поэтому когда мы вставляем линзы в оптическую модель, они располагаются в порядке, соответствующем их положению вдоль оптической скамьи. Таким образом, при реализации метода TraceRays мы сначала должны разрушить все существующие образы и лучи, а затем повторить все заново, используя список линз модели. Кроме того, мы должны сделать недействительным весь объект отображения, потому что мы полностью изменили его изображение:

```
procedure TOpticsModel.TraceRays;
var
  Image : TImage;
procedure ProcessLens (Item : TLens);
...
```

```

begin
    fImages.FreeAll;
    fRays.FreeAll;
    Image := fRealObject;
    EachLens (ProcessLens);
    if (fView <> nil) then
        begin
            fView.GetWindow.ForceRedraw;
            if fView.Focus then;
        end;
end;

```

Так как мы разложили алгоритм на небольшие части, локально вложенная процедура `ProcessLens` выполняет только небольшую часть работы. Она должна создать основные лучи, которые проходят от источника через линзы, и затем создать объект отображения в точке пересечения лучей. Если данная точка находится с той же стороны линзы, что и источник изображения, мы имеем мнимое изображение; если она находится с другой стороны линзы, мы имеем действительное изображение. Выше мы спроектировали класс `TRealOrVirtualImage`, чтобы действительное и мнимое изображения рисовались по-разному согласно их позиции. Таким образом, мы можем завершить вложенную процедуру следующим образом:

```

procedure ProcessLens (Item : TLens);
var
    Lens      : TLens;
    Rays      : TRays;
    RealOrVirtualImage : IRealOrVirtualImage;
begin
    Lens := Item;
    if (Lens.fFocalLength <> 0) then
        begin
            new (Rays);
            Rays.IShape (self);
            Rays.SetPoint (Image, Lens);
            fRays.InsertFirst (Rays);
            new (RealOrVirtualImage);
            RealOrVirtualImage.IShape (self);
            RealOrVirtualImage.SetPosition (Rays.ImageAt, Lens);
            fImage.InsertFirst (RealOrVirtualImage);
            Image := RealOrVirtualImage;
        end;
end;

```

Рассмотрим теперь класс `TRays`, так как именно здесь осуществляется реальное действие. Метод `SetPoint` должен определять точку прохождения лучей через фокус. Для этого мы должны обработать только два из трех основных лучей, так как в евклидовой геометрии две непараллельные линии в одной плоскости пересекаются в точке. Наиболее простой луч — луч, который проходит без изменения от источника через центр линзы, и другой простой луч — луч, который параллелен оптической оси и преломляется по направлению к точке фокуса.

Профессиональный программист, вероятно, уже должен был бы вовсе писать исходные тексты программ, но прервемся на момент и посмотрим на данную проблему с точки зрения объектно-ориентированной перспективы. Имеются ли какие-нибудь классы или объекты, которые интересуют нас здесь? Ответом будет «да». Мы объяснили наш алгоритм в терминах линий,

н, таким образом, имеет смысл ввести класс, который отражает данный взгляд на мир.

Мы можем проще всего определить пересечение двух лучей, если знаем уравнения линий, а уравнение каждой линии может быть задано двумя точками. Это позволяет иметь несколько операций над объектами-линиями, которые могут быть выражены в объявлении класса следующим образом:

```
TLine = object (TObject)

    fSlope : Real;
    fDelta : Real;

    procedure TLine.ILine (Point1, Point2 : Point);

    function TLine.Y (X : Real) : Real;
    function TLine.Intersection (ALine : TLine) : Point;

end;
```

Операции ILine и Intersection выполняются так, как это подсказывают их имена. Функция Y, имея параметром координату X, возвращает соответствующее значение вдоль оси Y согласно уравнению линии.

Мы можем поместить данный класс и его реализацию в тело модуля UOpticsModels потому, что нет необходимости в том, чтобы он был видимым другим классам или объектам в системе. После того как мы уже построили интерфейс данного класса, это легко завершить, используя евклидову геометрию:

```
procedure TLine.ILine (Point1, Point2 : Point);
begin
    fSlope := (Point1.V - Point2.V) / (Point1.H - Point2.H);
    fDelta := (Point1.V * 1.0) - (Point1.H * fSlope);
end;

function TLine.Y (X : Real) : Real;
begin
    Y := (fSlope * X) + fDelta;
end;

function TLine.Intersection (ALine : TLine) : Point;
var
    X      : Real;
    APoint : Point;
begin
    if (fSlope = ALine.fSlope) then
        SetPt (APoint, 0, 0)
    else
        begin
            X := (ALine.fDelta - fDelta) / (fSlope - ALine.fSlope);
            APoint.H := trunc(X);
            APoint.V := trunc(Y(X));
        end;
        Intersection := APoint;
end;
```

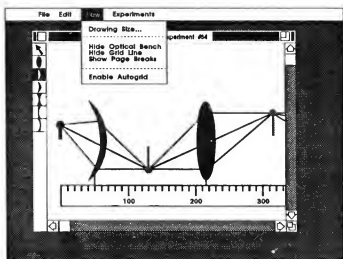


Рис. 9-17. Инструментальное средство разработки конструкций геометрической оптики.

Завершение метода `SetPoint` для класса `TRays` становится почти тривиальным: мы создаем линии для двух основных лучей и затем определяем их пересечение:

```
procedure TRays.SetPoint (FromImage : TImage; ThroughLens : TLens);
var
    FirstLine      : TLine;
    SecondLine     : TLine;
    APoint         : Point;
    SecondFocalPoint : Point;
begin
    fFromImage := FromImage;
    fThroughLens := ThroughLens;
    fLensCenter := fThroughLens.fCenter;
    new (FirstLine);
    FirstLine.lLine (fFromImage.fTop, fThroughLens.fCenter);
    SetPt (APoint,
           fThroughLens.fCenter.H, fFromImage.fTop.V);
    SetPt (SecondFocalPoint,
           fThroughLens.fCenter.H + fThroughLens.fFocalLength, kOpticalAxis);
    new (SecondLine);
    SecondLine.lLine (APoint, SecondFocalPoint);
    fToPoint := FirstLine.Intersection (SecondLine);
end;
```

Таким образом объектно-ориентированное проектирование является рекурсивным процессом. На примере класса `TLine` видно, как, углубляясь в реализацию, мы часто обнаруживаем новые обобщенные классы, которые имеют слабую связь со словарем предметной области, но представляют важные элементы механизмов, в которых мы нуждаемся для удовлетворения

функциональных требований. Одно важное достоинство данного рекурсивного процесса заключается в том, что мы часто завершаем разработку созданием классов, которые можно использовать в других приложениях; таким образом, объединение программных компонент многократного использования расширяется со временем.

Итак, реализация инструментального средства разработки конструкций геометрической оптики на данном этапе завершена. На рис. 9-17 показан снимок экрана рабочей версии приложения.

9.4. МОДИФИКАЦИЯ

Добавление новых возможностей

Отличительной чертой хорошо структурированной сложной системы является восприимчивость к изменениям. Ниже рассмотрены три функциональных усовершенствования инструментального средства разработки конструкций геометрической оптики и показано, как проект реагирует на эти изменения.

Первое усовершенствование касается производительности метода TraceRays. Можно увеличить скорость работы? Мы могли бы использовать гистограммные сервисные программы, которые указывают на программы, требующие для своего выполнения наибольшего времени. Из наблюдений за работающей системой следует, что программа тратит много времени на перерисовку оптического объекта отображения после прохождения луча. Это не считается чем-то необычным, так как операции рисования довольно сложные. Тем не менее, если мы вернемся к реализации метода TraceRays, мы увидим, что код исходного текста объекта отображения заставляет перерисовывать его целиком, даже если только часть объекта отображения подвергается воздействию. Поэтому одно очевидное усовершенствование заключается в оптимизации процесса изображения путем перерисовывания только тех областей объекта отображения, которые изменяются.

Это означает, что мы должны только слегка изменить реализацию метода TraceRays. Для этого сначала, прежде чем освободить изображение или луч, мы должны сообщить каждому объекту, чтобы он сделался недействительным. Затем вложенная процедура ProcessLens делает недействительными новые объекты: изображение и луч сразу после того, как они созданы и инициализированы. И наконец, мы можем убрать вызов метода ForceRedraw, так как другие изменения гарантированно аккумулируют области, включающие только измененные части объекта отображения, и эти изменения заставляют объект отображения производить новый проход луча не так резко, как другим способом. Итак, используя наши механизмы мы могли повысить производительность системы, оптимизируя только ключевую абстракцию. Более того, в начале разработки нашего проекта мы не задавались вопросами производительности, но предполагали, что эти вопросы будет легче решить после того, как мы будем иметь основу структуры системы, которую можно использовать в дальнейшем.

Предположим, мы убедили автора требований, что ограничения на приложение, связанные с использованием более четырех документов одновременно, были довольно произвольными. Для того чтобы ослабить эти ограничения, нам надо только изменить реализацию метода TOpticsDocumentManager. В частности, надо изменить представление класса, чтобы использовать список, а не массив документов. Затем надо модифицировать несколько мето-

дов, чтобы использовать данный список вместо массива. И наконец, мы должны изменить ресурс, который определяет Experiments-меню, так чтобы менеджер оптического документа мог добавлять или убирать элементы меню.

Поэтому здесь мы внесли изменения в требования, которые упрощают использование данного приложения. Внесение данного изменения требует модификации не одного метода; но поскольку мы использовали только один класс для инкапсуляции всего поведения, необходимого для управления множеством документов, и затем представляли только абстрактное поведение класса, данное изменение не воздействует на семантику любого другого класса. Нам надо выполнить recompilation некоторых программных модулей высокого уровня, но данное изменение не уменьшит нашу уверенность в корректности всех других классов.

Рассмотрим еще одно усовершенствование. Предположим, что на ранней стадии разработки от пользователей получен сигнал, что они хотят иметь на экране изображение точки фокуса линзы и данный атрибут должен переключиться подобно тому, как это выполняет команда меню ShowGridLines. Данное требование приводит к необходимости изменить реализацию более чем одного класса.

Начиная с самого нижнего программного модуля в нашем проекте, мы должны добавить новый параметр в методы DrawLens и Draw в классе TLens, указывающий, должна ли точка фокуса изображаться на экране. Данный параметр подобен существующему параметру DrawPosition. Мы должны внести подобные изменения в метод Draw класса TOpticalBench. Состояние данной команды переключения будет содержаться в объекте отображения так же, как и поле fDrawPageBreaks. В классе TOpticsView мы должны модифицировать метод Draw для использования данного нового поля. Нам также надо модифицировать методы DoSetUpMenus и DoMenuCommand в классе TOpticsView для обработки данной новой команды. Наконец, мы должны добавить данный элемент меню в наш ресурсный файл.

Внимательный читатель обнаружит, что мы внесли три различных типа изменений в наш существующий проект и эти изменения типичны для объектно-ориентированных систем. В первом случае мы модифицировали реализацию единственного метода, во втором — реализацию единственного класса, в третьем — интерфейсы и реализацию нескольких классов. Во всех этих классах изменения являются совместными. Другими словами, мы можем простыми средствами дополнить наши существующие механизмы и интерфейсы, для того чтобы изменить поведение системы. Нам не пришлось изобретать какие-либо новые механизмы или изменять основную семантику каких-либо ключевых абстракций. Мы убеждены, что наш проект инструментального средства разработки конструкций геометрической оптики является гибким; он может легко совмещать множество изменений в проблемной области.

Изменение требований

Скептически настроенный читатель может сказать, что мы выбрали только те изменения, которые легко реализовать. Рассмотрим изменение основных требований и посмотрим, как это повлияет на наш проект.

В ходе работы мы имеем дело только с частью проблем геометрической оптики — с явлениями преломления. Что произойдет, если мы включим в нашу проблемную область явления отражения? Это означает, что наше приложение должно было бы оперировать не только линзами, но и зеркалами.

Для того чтобы сделать нашу проблему более интересной, рассмотрим как плоские поверхности, так и сферические. Рассмотрим эти изменения снизу вверх. Анализ проблемы указывает на то, что мы должны согласовать три новых типа объектов: плоские, поверхности, выпуклые поверхности. Очевидным местом для размещения этих классов является программный модуль `UOpticsModels`; мы должны так модифицировать реализацию класса `TOpticsModel`, чтобы он содержал эти линзы как часть его состояния. Для этого мы спроектировали бы класс `TMirror` так, чтобы он имел те же суперклассы, что и класс `TLens`.

Что можно сказать о методе `TraceRays`? Существующий класс `TRays` не окажет действительной помощи, потому что он только содержит информацию об основных лучах, которые проходят через линзу. Тем не менее мы можем использовать его как образец, и мы можем, несомненно, рассчитывать на класс `TLine`. Таким образом, наша стратегия заключалась бы в реорганизации структуры классов; в результате появятся два класса: `TLensPrincipleRays` и `TMirrorPrincipleRays`, которые являются подклассами класса `TPrincipleRays`. Из-за явного разделения понятий среди этих абстракций реализация данных классов не является сложной — только немного утомительной.

Двигаясь вверх по архитектуре модулей, нам необходимо было бы сделать полезные изменения в различных классах объектов-команд. Действительно, логика существования классов команд такова, что объекты-команды мало заботятся об объектах, которыми они управляют. Итак, общее поведение линз и зеркал может быть собрано в общий подкласс, и затем мы можем определить все классы команд так, чтобы они основывались на данном общем классе. Это является примером сильных свойств полиморфизма.

Если мы будем перемещаться вверх по иерархии к программным модулям более высокого уровня, то изменения становятся менее существенными. Мы должны добавить несколько новых диалогов и новых команд (таких, как `Show Mirror Specification...`), но в других случаях классы объектов отображения, документов и приложения, которые мы уже определили, не зависят от изменений и расширений в классах нижнего уровня.

Новые требования на отражающие поверхности действительно приводят к изменениям в нашей существующей реализации. Но никакие из этих изменений не разрушают структуры нашего проекта.

Дополнительная литература

Фундаментальные основы геометрической оптики изложены *Sears, Zemansky и Young [1, 1987]*. В работе *Foley и van Dam [C, 1982]* обсуждается применение принципов геометрической оптики для алгоритмов прохождения лучей в компьютерной графике. Библиотека классов *MacApp* для *Object Pascal* описана в *MacApp [C, 1989]*. *Schmucker [C, 1986]* представляет введение в *MacApp*. Обобщения по языку программирования *Object Pascal* и примеры представлены в приложении. В библиографии предлагается несколько ссылок на различные системы, основанные на окнах, и объектно-ориентированные интерфейсы пользователя (см. раздел К, «Tools and Environments»).

Глава 10

С++. Система регистрации ошибок в программных средствах

Для большинства коммерческих приложений фирмы предпочитают использовать готовые системы управления базами данных (СУБД), гарантирующие реализацию параллельного доступа к данным, интеграцию, защиту и восстановление данных. Но любая СУБД требует обеспечения адаптации к задачам конкретного предприятия, поэтому в большинстве случаев создание подобных систем осуществляется в два этапа:

- 1) проектирование структуры базы данных экспертами по СУБД;
- 2) проектирование программных процессов обработки данных специалистами в прикладной области.

Такой подход имеет определенные достоинства, но оставляет нерешенными некоторые важнейшие вопросы. Между проектировщиками СУБД и программистами существуют принципиальные расхождения, связанные с различием методологий и приемов работы.

Проектировщикам СУБД свойственно смотреть на мир как на совокупность устойчивых и монолитных таблиц данных, а программисты видят мир в форме потоков управления. Пока невозможно рассчитывать на достижение единого интегрированного подхода к проектированию сложных систем. Если в системе преобладает проблема обработки данных, мы должны быть готовы к достижению компромисса в логике СУБД без учета особенностей задачи одинаково неэффективно и бессмысленно. Аналогично решение прикладной задачи без учета требований СУБД приводит к неразумным требованиям и к серьезным проблемам интеграции, обусловленным избыточными структурами данных.

В данной главе рассмотрена система обработки информации с целью показать возможность единого подхода к созданию СУБД и прикладной системы на основе методологии объектно-ориентированного проектирования. Для реализации системы использован язык С++, но приемы в значительной степени применимы и для обычных языков программирования (таких, как COBOL).

10.1. АНАЛИЗ

Определение границ задачи

Выше приведены требования к системе регистрации ошибок с очень малым объемом вычислений и очень большим объемом данных, которые требуется хранить, находить и перемещать. В процессе реализации проекта нам придется оперировать декларативными знаниями в гораздо большей степени, чем процедурными. Основа проекта — центральный вопрос объектно-ориентированного проектирования: какие ключевые абстракции характеризуют словарь предметной области и каковы механизмы управления ими?

Требования к системе регистрации ошибок

Программные средства с большим числом ошибок имеют мало шансов на долгую жизнь, что заставляет производителей программного обеспечения вести учет обнаруженных ошибок. Задача усложняется, если фирма ответственна за сопровождение различных программных средств с множеством версий и модификаций. Ошибки в программах могут обнаруживаться пользователями программ, персоналом, осуществляющим сопровождение, группой тестирования и интеграции, контролирующим персоналом или программистами. В случае обнаружения ошибка в первую очередь квалифицируется (ошибка в программе, неясность документации или простое заблуждение пользователя?), затем определяются ответственные за ее появление и приступают к устранению (фиксируют, ликвидируют или определяют как невоспроизводимую). Анализ ошибок помогает руководителям групп программистов правильно распределить ресурсы и оценить готовность программного продукта.

Наша задача состоит в создании системы регистрации ошибок для фирмы, разрабатывающей программные средства [1]. Такая система должна быть рассчитана на наличие многообразия программных продуктов и множественность версий для каждого продукта. Поскольку номенклатура продукции постоянно изменяется, система должна учитывать и изменения требований к СУБД.

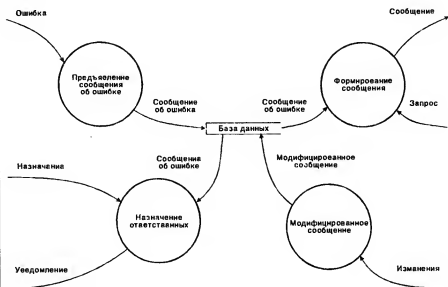


Рис. 10-1. Диаграмма потоков данных в системе регистрации ошибок.

На рис. 10-1 показаны основные потоки данных в такой системе. Сообщения об ошибках представляются для ознакомления и затем регистрируются в центральной базе данных. На рис. 10-1 не отражена множественность возможных источников сообщений об ошибках и различных способов их представления (звуковой, печатный и электронный). Для обнаруженной ошибки может быть определен ответственный. В этом случае соответствующим лицам передается уведомление об ошибке. Обычно сведения об ошибках, получаемые от пользователей, передаются в специальное подразделение (центр сопровождения) для квалификации. Большой процент ошибок здесь отсеивается, поскольку ошибки пользователей встречаются гораздо чаще, чем ошибки в программе. Если установлена ошибка в продукте, об этом сообщают. Затем выявляется причина ошибки и ее устраняют. Информация об ошибке корректируется по мере установления ее причин, например проверяющий повторил ситуацию возникновения ошибки и установил дополнительную информацию. Разработчик программы модифицирует код программы и устраняет ошибку. Множество людей на всех этих стадиях имеют доступ к базе данных. В частности, пользователь продукта может обратиться с запросом о состоянии дел по конкретной ошибке. Руководитель проекта может запросить отчет с перечнем зарегистрированных ошибок для анализа продукции. Группа контроля и интеграции программного продукта при выпуске новой версии может обратиться к базе данных по вопросу зафиксированных и устраненных ошибок.

Для простоты будем полагать, что наша система реализует текстовый интерфейс пользователя, хотя в последующем интерфейс можно будет усложнить. Кроме того, допустим, что представление сведений об ошибках осуществляется только в форме сообщений на дисплее. По ряду причин целесообразно ограничиться одной стандартной СУБД для хранения данных об ошибках. Однако большое число потенциальных пользователей требует обеспечения иллюзии распределенной базы данных.

По природе подобные системы являются открытыми для расширения. В процессе анализа мы подойдем к пониманию того, какие ключевые абстракции существенны для организации в конкретное время: определим виды данных, формы сообщений, запросы на обработку и другие аспекты, вытекающие из задач, решаемых предприятием. Мы употребляем фразу «в конкретное время», поскольку бизнес не стоит на месте. Он подвержен изменениям рынка, а система обработки информации должна учитывать эти изменения. Устаревшие программные продукты становятся экономически невыгодными. Поэтому мы должны подойти к проектированию системы с учетом ее последующей модернизации. Опыт подсказывает, что вероятнее всего изменятся два элемента нашей системы:

- * Типы обрабатываемых данных.
- * Аппаратные средства реализации системы.

С течением времени появятся новые виды продукции и новые пользователи, а часть старых видов продукции будет снята с производства и эксплуатации. При практическом использовании системы может возникнуть необходимость в получении дополнительной информации об ошибке (твердые копии графических изображений на экране и даже звуковые фрагменты). Новые аппаратные средства разрабатываются быстрее, чем программные системы, и компьютеры устаревают за несколько лет. Однако невозможно так

быстро заменять сложные программные системы. Затраты на разработку программных систем часто существенно превышают стоимость оборудования, а риск от внедрения новой системы слишком велик, чтобы предприятия в повседневной работе могли позволить себе менять используемые программные продукты. Из этого следует вывод о необходимости со временем менять интерфейс пользователя. Для большинства систем обработки информации простой текстовый или экранный интерфейс вполне удовлетворителен. Однако удешевление компьютеров и совершенствование графических возможностей являются предпосылкой для модернизации интерфейсной методологии. В отличие от системы обработки информации в системе регистрации ошибок интерфейс пользователя не играет такой важной роли. Ядром рассматриваемой системы является база данных, а интерфейс пользователя представляет собой оболочку этого ядра. Вполне реально (и очень желательно) сделать интерфейс пользователя открытым для последующих изменений. Для пользователей, работающих с сообщениями об ошибках, вполне достаточен текстовый интерфейс. Печать отчетов на бумаге может осуществляться в пакетной форме, хотя отдельные руководители проектов могут обратиться к графическому интерфейсу в интерактивном режиме. Цель нашего проекта не требует глубокой проработки этого вопроса, мы лишь упоминаем о возможных видах интерфейса.

Сформулируем два стратегических проектных решения. Во-первых, мы будем строить систему на основе одной стандартной СУБД. Специальная разработка СУБД в нашей ситуации не имеет смысла, поскольку потребует больших дополнительных затрат и не гарантирует нужной гибкости системы. Кроме того, стандартные коммерческие СУБД обладают преимуществом и могут применяться на широком спектре аппаратных средств (от персональных компьютеров до суперЭВМ). Во-вторых, мы будем использовать распределенную сеть для реализации проектируемой системы. Для простоты предположим, что база данных размещена на одном компьютере, но доступ к ней возможен со стороны множества других компьютеров. Такое проектное решение называется методом клиент/сервер, когда компьютер с базой данных выполняет роль сервера для множества внешних клиентов. Для сервера не играют роли конкретные особенности компьютеров клиентов, даже если это компьютер со своей локальной базой данных. Такой подход позволяет создать систему регистрации в виде неоднородной сети, что делает эту систему мало чувствительной к изменениям типа используемых аппаратных средств реализации.

Реляционная модель базы данных

Модели баз данных. По Дейту, база данных — «это среда для хранения данных, обладающая свойствами единства и разделения ресурсов. Под единством подразумевается возможность рассмотрения всех файлов базы данных как единой структуры с минимальной избыточностью. Под разделением ресурсов понимается возможность работы нескольких пользователей с любыми фрагментами базы данных» [2]. Централизация управления базой данных «позволяет разрешить возникающие коллизии, стандартизировать механизмы, обеспечить защиту данных и общность управления» [3].

Проектирование базы данных представляет собой достаточно трудную задачу, поскольку приходится удовлетворять противоречивые требования. Необходимо обеспечить не только функциональные свойства базы данных, но учесть факторы быстродействия и размеры используемой памяти. База дан-

ных, которая слишком долго ищет требуемые данные, становится практически бесполезной, так же как функциональная неэффективность поглощает все ресурсы компьютера или требует многочисленного обслуживающего персонала.

Проектирование базы данных имеет много общего с процессом объектно-ориентированного проектирования. Методология баз данных рассматривает проектирование как последовательно-итеративный процесс, требующий принятия как логических, так и физических решений [4]. Внорковски и Кул называют «объекты, характеризующие базу данных, с точки зрения пользователей и программистов, логическими. Физические объекты характеризуют способы фактического хранения данных в системе» [5]. В отличие от методологии ООП разработчикам базы данных приходится постоянно в процессе проектирования совершать переходы от физической структуры к логической и обратно. Способы описания элементов базы данных мало отличаются от описания ключевых абстракций в ООП. Проектировщики СУБД для анализа принимаемых решений используют диаграммы соотношений между объектами. Как мы ниже увидим, диаграммы классов не только отвечают требованию отражения таких соотношений, но и обладают даже большей выразительностью. Действительно, предположим, что при создании любой базы данных нужно ответить на следующий вопрос: «Какие структуры данных и соответствующие им операции должна поддерживать система?» [6]. В зависимости от ответа на этот вопрос базы данных относятся к одному из следующих трех классов (моделей):

- * Иерархические.
- * Сетевые.
- * Реляционные.

Позже был определен четвертый класс баз данных: объектно-ориентированные базы данных (ООБД). Преимущества ООБД были доказаны и подтверждены в таких областях, как компьютерная инженерия и компьютерная программная инженерия, где требуется обрабатывать большое количество разнородных данных.

Реляционная модель СУБД. Основу реляционных баз данных составляют «таблицы, колонки которых представляют данные и их атрибуты, а строки соответствуют конкретным значениям каждого элемента базы данных... Кроме того, имеется некоторый набор операторов, позволяющий манипулировать содержанием базы данных и извлекать из нее нужную информацию» [7]. Возьмем для примера фрагмент базы данных, представляющий собой опись элементов. Мы имеем здесь компоненты, идентифицируемые номером (PNumber), а также наименование компонент (PName), например, следующим образом:

Компоненты

PNumber	PName
0081735	Резистор, 100 Ом 1/4 Вт
0081736	Резистор, 100 Ом 1/4 Вт
3891043	Конденсатор, 100 пФ
9074000	Микросхема 7400
9074001	Микросхема 74LS00

В этой таблице две колонки представляют различные атрибуты. В данном конкретном случае порядок колонок и строк значения не имеет. Число строк может быть любым, но строки не должны повторяться. Заголовок PNumber является исходным ключом, который используется для идентификации записей.

Компоненты поступают со складов, которые можно охарактеризовать номером, наименованием, адресом и телефоном. Этому отвечает следующая форма записи:

Склады

SNumber	SName	SAddress	Telephone
00056	Interstate Supply	2222 Fanning, Amarillo, TX	806-555-0036
3107	Interstate Supply	3320 Scott, Santa Clara, CA	408-555-3600
78829	Universal Products	171 Parfet Ct, Lakewood, Co	303-555-2405

Исходным ключом здесь является SNumber, поскольку он однозначно определяет нужный склад. Все строки этой таблицы различаются, хотя в двух из них имеется одинаковое значение наименования склада.

Компоненты с разных складов могут поставляться по разным ценам, поэтому нам потребуется также таблица цен. Для каждого сочетания компонента/склад в таблице указывается цена:

Цены

PNumber	SNumber	Price
0081735	03107	\$0.10
0081735	78829	\$0.09
0156999	78829	\$367.75
7775098	03107	\$10.90
6889655	00056	\$0.09
9074001	03107	\$1.75

Эта таблица не имеет единственного исходного ключа. Для идентификации строк здесь следует использовать сочетание SNumber и PNumber. Такой ключ называется составным. Отметим, что в этой таблице нет имен компонент и складов, поскольку это было бы избыточно — нужную информацию можно найти в предыдущих таблицах. Ключи PNumber и SNumber называются чужие, так как это исходные ключи других таблиц.

Для ведения учета нам еще потребуется таблица с данными о количестве имеющихся на складе компонент:

Наличие

PNumber	Quantity
0081735	1000
0097890	2000
0156999	34
7775098	46
6889655	1
9074001	192

Можно было бы включить количество компонент отдельной колонкой в таблицу компонент. Однако мы этого не делаем, поскольку наличие компонент характеризует текущее состояние складов, а таблица компонент содержит всю номенклатуру используемых элементов. Это еще один пример подхода к проблеме схожести и различия в дополнение к примеру из гл. 4.

SQL. Пользователю необходимо иметь возможность выполнения некоторого набора стандартных операций над данными указанных таблиц. Возможно, придется включить в таблицы новые склады, удалить некоторые компоненты или изменить данные о наличии компонент. Следует также обеспечить доступ к данным таблиц. Например, мы можем затребовать список всех компонент по конкретному складу, а также список компонент, удовлетворяющих некоторому количественному критерию. Может, наконец, потребоваться исчерпывающий отчет с указанием стоимости заданного набора компонент по наиболее дешевому варианту. Такого рода операций реализуются почти во всех реляционных СУБД на языке SQL (язык структурных запросов). Язык SQL используется как в программном, так и в интерактивном режиме.

Основной конструкцией SQL является оператор выбора, существующий в следующих формах:

```
SELECT <attribute>
FROM <relation>
WHERE <condition>
```

Например, чтобы запросить номера компонент, для которых запас составляет менее 100 штук, следует записать

```
SELECT PART, QUANTITY
FROM INVENTORY
WHERE QUANTITY < 100
```

Возможны и более сложные запросы. Можно, например, вместо номеров компонент затребовать их наименования:

```
SELECT PNAME, QUANTITY
FROM INVENTORY, PARTS
WHERE QUANTITY < 100
AND INVENTORY.PART = PARTS.PNAME
```

Такой запрос называется соединением, поскольку в нем объединяются два и более условия в одно общее условие. Приведенный выше запрос не создает новую таблицу, а лишь возвращает некоторое число строк. Оператор выбора может возвращать сколь угодно большое число строк и нужно иметь возможность просматривать их последовательно. В языке SQL для этого используется механизм курсора, аналогичный операторам итерации, изложенным в гл. 3. Курсор можно определить следующим образом:

```
DECLARE C CURSOR
  FOR SELECT PNAME, QUANTITY
    FROM INVENTORY, PARTS
   WHERE QUANTITY < 100
   AND INVENTORY.PART = PARTS.PNAME
```

Для реализации этого соединения нужно записать следующую строку:

```
OPEN C
```

Чтобы просмотреть соединение построчно, записываем следующий оператор:

```
FETCH C INTO NAME, AMOUNT
```

После просмотра курсор закрывается командой

```
CLOSE C
```

Вместо курсора можно сформировать фактическую таблицу с результатами выполнения запроса. Такая таблица называется обзором и может использоваться для дальнейших операций наряду с другими таблицами. Напим, например, операторы создания обзора содержащего наименования компонент, складов, а также стоимость:

```
CREATE VIEW V (PNAME, SNAME, COST)
  AS SELECT PARTS.PNAME, SUPPLIERS.SNAME, PRICES.PRICE
    FROM PARTS, SUPPLIERS, PRICES
   WHERE PARTS.PNUMBER=PRICES.PNUMBER
   AND SUPPLIERS.SNUMBER=PRICES.SNUMBER
```

Обзоры позволяют различным пользователям получить нужную именно им информацию из базы данных. Обзоры могут быть независимы от модели базы данных и тем самым создают значительную степень свободы. С помощью механизма «обзор по обзору» осуществляется защита данных базы данных и гарантируется контроль доступа к данным. Отличие обзоров от обычных таблиц базы данных состоит в том, что их нельзя произвольно изменить.

Для решаемой нами задачи язык SQL является абстракцией низкого уровня. Мы не можем требовать от пользователей системы знания SQL, и этот язык не является элементом словаря предметной области. SQL мы будем использовать только в реализации наших процедур внутри системы, скрыв этот язык от пользователя системы.

Анализ базы данных

«Если задана основа представления данных в базе данных, то как определить логическую структуру, приемлемую для этих данных? Другими словами, как выбрать нужные соотношения и их атрибуты? Это и является сутью проектирования базы данных» [8]. Оказывается, что определение ключевых абстракций базы данных — процесс, аналогичный определению классов и объектов в методологии ООП.

Нормализация. Наиболее важной целью проектирования базы данных является хранение каждого факта в одном месте. Это снижает избыточность информации, упрощает процесс модификации данных, обеспечивает целостность базы данных (внутреннюю устойчивость и непротиворечивость) и снижает объем занимаемой памяти. Достичь этой цели не так легко (и, как выясняется, не всегда обязательно).

Для достижения этой цели разработана теория нормализации (хотя это не единственный подход [9]). Нормализация характеризует свойство таблиц удовлетворять ряду требований (иметь нормальную форму). Нормальные формы имеют несколько уровней, каждый из которых основывается на предыдущих [10]:

- | | |
|---------------------------------|--|
| * Первая нормальная форма (1NF) | Каждый атрибут является простым значением (не распадается на другие атрибуты) |
| * Вторая нормальная форма (2NF) | Таблица соответствует 1NF и каждый атрибут определяется своим ключом (атрибуты функционально независимы) |
| * Третья нормальная форма (3NF) | Таблица соответствует 2NF и ни один атрибут не содержит фактов о других атрибутах (совместная независимость атрибутов) |

Таблицы, соответствующие 3NF, «содержат свойства ключей, всех ключей и ничего, кроме ключей» [11].

Все приведенные выше таблицы имеют форму 3NF. Существуют и более высокие уровни нормализации (главным образом в отношении многомерных фактов), но для нашей цели они не существенны. В SQL существует несколько практических ограничений, которые приводят к тому, что нормализация не может быть единственным критерием проектирования.

В частности, SQL имеет очень ограниченный набор типов данных (символы, строки фиксированной длины, целые числа, действительные числа). Практические задачи часто выходят за рамки такого ограниченного набора типов. Невозможно непосредственно представить в базе данных изображения или большие фрагменты текста. Кроме того, SQL не позволяет модифицировать обзоры, полученные в процессе сложных запросов-соединений.

Анализ предметной области для системы регистрации ошибок. Сначала сформируем список ключевых абстракций, а затем создадим из них нормализованные таблицы. Сообщения об ошибках всегда кем-то инициализируются (предъявляются). Мы можем проследить источник возникновения сообщений, если будем иметь следующую информацию:

- * Имя предъявителя.
- * Телефон предъявителя.
- * Идентификатор клиента (ID).

Каждый клиент имеет свой идентификатор ID в соответствии с лицензией на использование конкретной версии продукта. Регистрация (ID) позволяет контролировать права предъявителя на использование продукта и на оказание услуг. Для регистрации необходимы следующие данные:

- * Дата регистрации.
- * Идентификатор версии продукта.
- * Идентификатор клиента.

Один клиент может быть пользователем нескольких видов продукции. Более того, сообщение об ошибке могут передать представители предприятия пользователя продукта. Для нас важно убедиться в связи инициатора с предприятием-клиентом. Это позволяет выявить случаи «пиратского» использования копий программных продуктов. Нам необходима следующая ключевая информация о клиенте:

- * Наименование фирмы.
- * Адрес фирмы.

Каждый продукт может существовать в нескольких одновременно используемых версиях. Поэтому мы должны иметь следующие данные по каждой версии продукта:

- * Идентификатор продукта.
- * Сведения о версии.

Кроме того, нужна дополнительная информация о каждом продукте:

- * Наименование продукта.
- * Перечень ответственных групп.

Под группами ответственных подразумевается следующее: каждый вид продукции сопровождается конкретной командой (командами) разработчиков и специалистов по эксплуатации. Именно эти команды должны получить сообщения об обнаруженных в продукции ошибках. Каждая группа ответственных имеет свой номер, наименование и руководителя, поэтому отметим и эти данные:

- * Идентификатор группы.
- * Наименование группы.
- * Руководитель группы.

Группы состоят из сотрудников компании, информация о которых также существенна для нас:

- * Идентификатор сотрудника.
- * Имя сотрудника.
- * Почтовые атрибуты сотрудника.

Почтовые атрибуты сотрудника включают, в отличие от полного имени адрес электронной почты. Сотрудник может входить в несколько групп и быть программистом, специалистом по эксплуатации, обслуживанию и маркетингу. Для клиентов важно знать координаты специалистов по обслуживанию и маркетингу. Поэтому в информацию клиента включим следующие сведения:

- * Представитель по обслуживанию.
- * Представитель по маркетингу.

Инициатор (предъявитель) сообщает об обнаружении ошибки в конкретной версии продукта. Наряду с идентификатором ошибки нам необходима следующая информация:

- * Идентификатор предъявителя.
- * Дата предъявления.
- * Приоритет предъявителя.
- * Идентификатор версии продукта.

- * Вид ошибки.
- * Подробное описание ошибки.

Подробное описание может иметь различную форму: текст, твердая копия изображения, рисунок и т.д. Устанавливается группа, ответственная за ошибку, а затем конкретный исполнитель. Исполнитель может изменить следующие данные:

- * Внутренний приоритет.
- * Текущее состояние.

Состояние характеризует стадию, на которой находится решение конкретного вопроса:

- * Еще не рассматривается.
- * Анализ причин.
- * В процессе устранения.
- * Устранено.
- * Отклонено.
- * Ошибка отсутствует.
- * Ошибка невозпроизводима.
- * Отсутствие действий.

В сообщении может упоминаться несколько действий по данной ошибке. Каждое действие включает анализ, комментарий, тест или объяснения отвечающего за устранение ошибки. Действия по сообщению содержат:

- * Описание предпринятых действий.
- * Ответственный.
- * Дата проведения.
- * Идентификатор сообщения.

При завершении действий по данному сообщению необходимо отразить итоговое состояние (факт устранения) и указать другие версии продуктов, которые могут содержать ту же ошибку. Это позволит в последующих версиях учесть обнаруженные ошибки на основе данных:

- * Выполненные действия.
- * Установленные ошибки.

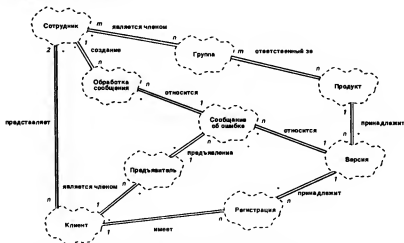


Рис. 10-2. Диаграмма классов системы регистрации.

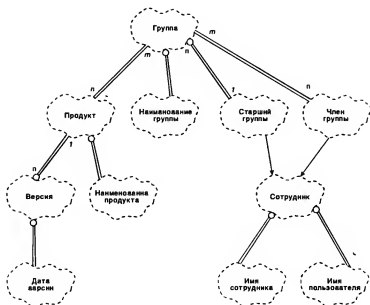


Рис. 10-3. Продукты, версии, группы и сотрудники.

И наконец, добавим еще фрагмент информации:

• Аудиторы.

Это перечень лиц, которые должны быть уведомлены о произведенных действиях.

Схема системы регистрации ошибок. Мы сформулировали перечень абстракций, отвечающих сути поставленной задачи, но не упорядочили его. Для нормализации базы данных необходимо установить иерархию указанных видов данных.

Начнем с самых общих групп данных. Анализ показывает наличие девяти таких групп (кластеров): сообщения об ошибках, сообщения о действиях по выявлению причин, программные продукты, версии продуктов, группы, сотрудники, клиенты, предъявители и регистрация. Эти проектные решения приведены на рис. 10-2.

Диаграмма классов не случайно похожа на схему отношений объектов. На ней отмечены количественные характеристики отношений и сделаны некоторые пояснения (точкой показано направление, к которому относится та или другая метка). Из рисунка видно, что группа отвечает за несколько продуктов, которые могут иметь версии. Предъявитель (сотрудник фирмы) может инициировать сообщения об ошибках в произвольном количестве по каждой версии.

На рис. 10-3 показаны существенные связи между продуктами, версиями, группами и сотрудниками. Простые связи между классами означают отношения использования. Введены также две промежуточные абстракции — руководитель группы и член группы — для различения уровня сотрудников.

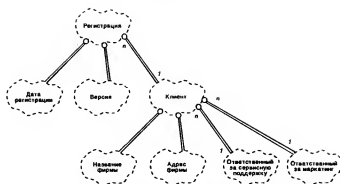


Рис. 10-4. Регистрация и клиенты.

Из данной диаграммы вытекает структура базы данных:

PRODUCTS

product ID
product name

RELEASE

release ID
product ID
release date

GROUPS

group ID
group name
group leader

EMPLOYEES

employee ID
employee name
employee user name

Все таблицы имеют форму 3NF.

Отношения между группами и продуктами, между группами и сотрудниками не определены, поскольку такие связи не имеют практического смысла. Продукт может включать перечень ответственных, и, наоборот, данные о группе могут содержать перечень продуктов, связанных с этой группой. Такие неоднозначные соотношения (1:n и m:n) могут вносить избыточность, если их неправильно определить. Для определенности введем две вспомогательные таблицы:

PRODUCT RESPONSIBILITIES

product ID
group ID

GROUP ASSIGNMENTS

group ID
employee ID

Две колонки в каждой таблице образуют составной ключ. На рис. 10-4 показана структура классов регистрации и клиентов, которую можно описать следующим образом:

REGISTRATIONS

registration	ID
registration	date
release	ID
customer	ID

CUSTOMERS

customer	ID
company	name
company	address
field-support	representative
marketing	representative

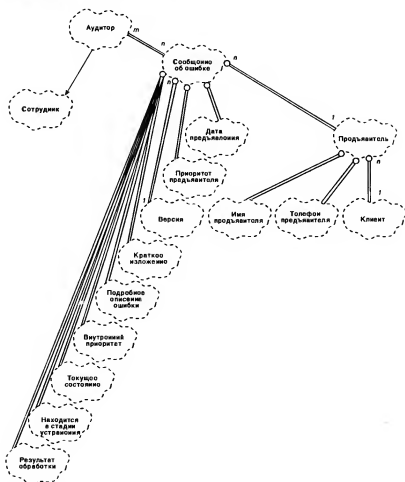


Рис. 10-5. Предъявители сообщений об ошибках, сообщения о действиях, изменения аудитором.



Рис. 10-6. Сообщения о действиях по выявлению причин ошибок.

Вспомогательные таблицы здесь не нужны, поскольку отношения полностью определены. Нам остается определить абстракции предъявителей сообщений об ошибках, сообщений о действиях и изменениях аудиторов. Эти решения приведены на рис. 10-5 и 10-6.

Из этих двух диаграмм можно составить следующие таблицы данных:

SUBMITTERS

submitter ID
submitter name
submitter telephone
customer ID

PROBLEM REPORTS

problem report ID
submitter ID
date submitted
submitter priority
release ID
problem summary
detailed problem description
Internal priority
current status
fixed in release
maintenance summary

AUDITORS

problem report ID
employee ID

MAINTENANCE REPORTS

maintenance report ID
maintenance report description
creator
date created
problem ID

Отметим, что мы ввели два подкласса сотрудников. Аудиторы — это сотрудники, которые должны уведомляться о любых изменениях в состоянии дел (например, о любом сообщении о действиях), а ответственные — это сотрудники, производящие действия по сообщениям.

Для устранения избыточности в базе данных все таблицы должны быть нормализованы. С этой точки зрения приведенные описания являются законченными (исключая форму описания) и позволяют администратору базы данных приступить к описанию схемы на языке SQL. Однако все, что мы сделали выше, представляет низкий уровень абстракции. Пользователям системы не обязательно знать о наличии и видах таблиц в базе знаний. Ниже мы рассмотрим проектирование системы на более высоком уровне абстракции.

10.2. ПРОЕКТИРОВАНИЕ

Архитектура процессов

По ряду причин мы приняли выше проектное решение, по которому система регистрации ошибок должна функционировать в распределенной сети. Из этого следует тот факт, что система регистрации не представляет собой одну программу, как в двух предыдущих примерах, а состоит из набора взаимодействующих программ, обеспечивающих функциональные свойства системы.

Что касается архитектуры процессов, мы прежде всего можем сделать вывод о различной роли пользователей в системе регистрации ошибок. Из предыдущего анализа можно выделить семь групп пользователей:

- * Предъявители (инициаторы сообщений об ошибках).
- * Ответственные (по устранению причин ошибок).
- * Аудиторы.
- * Руководители групп.
- * Члены групп.
- * Представители по обслуживанию.
- * Представители по маркетингу.

Анализируя роли этих групп пользователей, можно выделить следующий набор действий:

- * Предъявление сообщений об ошибках.
- * Обработка сообщений об ошибках.
- * Изменение сообщений об ошибках.
- * Формирование сообщений о предпринятых действиях.
- * Запросы в базу данных.

Просматривая этот перечень, можно заметить, что одна из ролей опущена. Любая база данных содержит огромное количество функций управления, таких, как архивирование данных, прием-передача данных из различных внешних источников, диагностические сообщения, измерение характеристик базы данных и т.д. Все эти действия осуществляются администратором базы данных. Большинство таких функций обеспечиваются стандартными СУБД, и мы не рассматриваем их реализацию, а лишь указываем на их важное значение.

Приведенный выше перечень действий (транзакций) соответствует базовым пользовательским функциям системы регистрации ошибок. Такие функции можно выполнить как в виде отдельных программ, так и в виде общей программы с возможностью выбора пользователем конкретной транзакции. В любом случае необходимо определить место реализации этих транзакций. Одним из решений является возможность реализации любых функций на любом компьютере в сети, но за такую универсальность всегда нужно платить, поэтому другим решением является специализация — закрепление

каждой транзакции за отдельным компьютером. В нашем случае решено использовать стандартную коммерческую СУБД с централизованной базой данных.

На рис. 10-7 показана диаграмма процессов, в которой реализуется компромиссное решение. Из рисунка видно, что коммерческая СУБД размещена на отдельном компьютере, который выполняет роль сервера сети. Этот компьютер может принадлежать любому классу, начиная от РС и кончая мощным компьютером. Перенос данных на различные ЭВМ гарантируется использованием стандартной СУБД. Этот же компьютер реализует все функции DBA. Все остальные компьютеры сети являются равноправными элементами системы в пределах решаемых ею задач. В результате мы получаем неоднородную сеть (сеть, включающую различные типы компьютеров), узлы которой не обязательно гарантируют выполнение всех видов транзакций (могут отсутствовать дисплейные и печатающие устройства).

Для обеспечения взаимодействия всех элементов системы независимо от вида сетевых средств мы должны создать обобщенный механизм связи. Наличие этого механизма отражено на рис. 10-7. Мы видим, что реализация транзакций на любом компьютере осуществляется путем взаимодействия с базой данных через механизм удаленного вызова процедур (Remote Procedure Call). Этот механизм позволяет написать для компьютера А программу, которая обращается к подпрограмме, написанной на компьютере В. Для передачи сообщений об ошибках будем использовать средства электронной почты. Электронная почта привлекательна тем, что является распространенным средством для разнородных сетей, а правила пользования ею известны большинству клиентов. Изложенный подход — типичный пример создания нового механизма на основе уже известных. Использование существующих средств (RPC и электронной почты) упрощает нашу задачу (уменьшает объем работы) и уменьшает степень риска (использует уже проверенные средства).

На основе рис. 10-7 мы можем строить дальнейшие заключения о функционировании системы. Например, о том, как клиенты предъявляют сообщения об ошибках. Предъявление сообщений — это скорее вопрос, относящийся к политике бизнеса, а не к технике. На практике допускается, чтобы пользователь продукта мог выбрать компьютер в сети и послать сообщение. Также можно пропустить все сообщения через специальный центральный узел обслуживания: клиенты сообщают туда письменно или по телефону об обнаруженных ошибках, а персонал центрального узла вводит эту информацию в систему регистрации. Выбор того или другого подхода не отражается на принимаемых технических решениях, так как мы создаем общий механизм, пригодный для любого варианта реализации.

Схема базы данных

Схему базы данных будем создавать, ориентируясь на язык C++. Определим, какие классы составляют внутреннюю структуру системы регистрации на уровне абстракции реляционной СУБД, не пользуясь словарем предметной области. Классы, приводимые в этом разделе (на языке C++), соответствуют схеме базы данных (изложенной на языке SQL).

Наша задача упрощается тем, что уже имеются решения для этих абстракций. На данном уровне абстракции классы сильно структурированы по назначению, пригодны в качестве буферов для приема-передачи фрагментов базы данных. Для этих классов не определяются специфические операции,

но в дальнейшем на основе таких классов будут строиться абстракции более высокого уровня, имеющие определенное поведение.

Причина низкого уровня абстракций состоит в низком уровне SQL. Уже говорилось, что в языке SQL набор типов данных очень ограничен. Поэтому классы низкого уровня на C++ только отражают типы SQL с учетом проблемной специфики всех видов отношений.

Просмотр всех таблиц, входящих в схему базы данных, позволяет выделить девять элементарных типов данных. Определим эти типы, являющиеся общими для всех пользователей системы:

```
typedef int      ID;
typedef char*    Name;
typedef char*    Address;
typedef int      Telephone;
typedef char*    Date;
typedef int      Priority;
typedef char*    Summary;
typedef char*    FileName;
enum Status     {Reviewing, InProgress, Fixed, Defer,
                 NotAProblem, NotReproducible, NoAction};
```

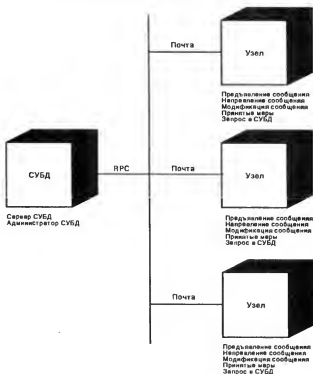


Рис. 10-7. Диаграмма процессов системы регистрации.

Большинство этих определений очевидны. Идентификаторы ID являются обычными числами, а имена и адреса (Names, Addresses) — символьными строками. Состояние (Status) включает семь пронумерованных значений, рассмотренных выше. Эти определения могут быть легко изменены, что не окажет серьезного влияния на другие проектные решения.

Определение типа FileName (имя файла) требует некоторого пояснения. Мы уже говорили об ограничениях SQL на типы атрибутов. SQL позволяет работать со строками переменной длины и двоичными кодами, но в большинстве реализаций размер данных ограничивается значениями порядка 256 байт и менее. Для нас такое ограничение является критическим, и мы должны его устранить косвенным путем. Для этого выберем файловый способ хранения и передачи длинных сообщений. Атрибут содержит только имя такого файла, а сам файл всегда может быть найден сетевыми средствами.

В качестве суперкласса для всех видов соотношений введем класс Relation, который будет определять все операции подклассов в процессе специализации суперкласса. Это гораздо удобнее, чем использование обычной структуры (structs) в языке C. Абстракции для продукции, версий, групп и сотрудников будут выглядеть следующим образом:

```
class LowLevelProduct : public Relation {
public:
```

```
    ID        ProductID;
    Name      ProductName;
```

```
};
```

```
class LowLevelRelease : public Relation {
public:
```

```
    ID        ReleaseID;
    ID        ProductID;
    Date      ReleaseDate;
```

```
};
```

```
class LowLevelGroup : public Relation {
public:
```

```
    ID        GroupID;
    Name      GroupName;
    ID        GroupLeader;
```

```
};
```

```
class LowLevelEmployee : public Relation {
public:
```

```
    ID        EmployeeID;
    Name      EmployeeName;
    Name      EmployeeUserName;
```

```
};
```

По причинам, которые скоро прояснятся, в названии каждого класса используется приставка LowLevel.

Классы ответственных за продукцию и ответственных групп состоят из объектов уже определенных классов:

```
class LowLevelProductResponsibility : public Relation {
public:
```

```
    ID        ProductID;
    ID        GroupID;
```

```
};
```



```
class LowLevelGroupAssignment : public Relation {
public:
    ID      GroupID;
    ID      EmployeeID;
};
```

Следующий класс будет выглядеть так:

```
class LowLevelProblemReport : public Relation {
public:
    ID      ProblemReportID;
    ID      Submitter;
    Date    DateSubmitted;
    Priority SubmitterPriority;
    ID      ReleaseID;
    Summary ProblemSummary;
    FileName DetailedProblemDescription;
    Priority InternalPriority
    Status  CurrentStatus;
    ID      FixedInRelease;
    Summary MaintenanceSummary;
};
```

Остальные определения классов нижнего уровня достаточно просты и для краткости опущены.

Механизм SQL

Все описанные выше классы представляют низкий уровень абстракции и отражают не столько предметную область, сколько механизмы внутренней реализации. Было бы желательно создать абстракции более высокого уровня, которые позволят обрабатывать запросы базы данных на уровне объектов, а не на уровне фрагментов различных таблиц. Действительно, любая транзакция должна выполнить операции с сообщениями как с неделимыми объектами, а содержание таблиц должно быть надежно защищено от случайного воздействия.

Построение абстракций на языке SQL. Механизм SQL иллюстрируется рис. 10-8, где показан сервер СУБД (реализованный на отдельном компьютере), выполняющий все операции низкого уровня по обработке сообщений. В узлах сети размещаются пользователи СУБД, взаимодействующие с сервером для осуществления транзакций системы регистрации. Все транзакции выполняются на более высоком уровне абстракций, а пара клиент-сервер отвечает за их реализацию в коде низкого уровня. Поскольку этот процесс связан с отображением запросов высокого уровня на языке SQL, то мы называем его механизмом SQL.

Классы низкого уровня для работы с базой данных уже определены нами и теперь необходимо перейти к разработке классов, отражающих предметную область системы регистрации. Мы выполним краткий анализ предметной области для всех видов транзакций и определим перечень операций, которые будут выполняться над объектами высокого уровня.

Из анализа определяются три исходных вида операций в системе:

- * Препъявление Первичное препъявление (иницирование) сообщения об ошибке.


```

virtual Submitter      SubmitterOfReport () const;
virtual Date           DateOfReport () const;
virtual Priority        SubmitterPriorityOfReport () const;
virtual Release         ReleaseWithProblem () const;
virtual Summary         SummaryOfProblem () const;
virtual Description     DescriptionOfProblem () const;
virtual Priority        InternalPriorityOfReport () const;
virtual Status          StatusOfReport () const;
virtual Release         ReleaseWithFix () const;
virtual Auditors        AuditorsOfReport () const;
virtual MaintenanceReports MaintenanceReportsForProblem () const;

```

protected:

```

Submitter      TheSubmitter;
Date           TheDate;
Priority        TheSubmitterPriority;
Release        TheProblemRelease;
Summary        TheSummary;
Description     TheDescription;
Priority        TheInternalPriority;
Status         TheStatus;
Release        TheFixedRelease;
Auditors       TheAuditors;
MaintenanceReports TheMaintenanceReports;

virtual int     SetProblemReportID (ID AnID);
virtual ID      IdOfProblemReport () const;

```

private:

```

ID      ProblemReportID;

```

};

Предполагается, что видимость классов и типов, входящих в данный класс, обеспечена (Release, Status, Priority и т.д.).

На языке C++ необходимо определить для каждого класса конструктор и деструктор, что сделано в приведенном описании. Структура класса защищена для всех пользователей, кроме подклассов данного класса (декларация protected:). Для обеспечения доступа к структуре данных этого класса определены операции-селекторы. Возможность переопределения функций в подклассах реализуется с помощью квалификатора virtual. Функции определяются в качестве фактически (виртуальных) всегда, когда нет явных причин отказаться от их последующего переопределения.

Особый случай представляет объект класса, обозначенный именем ProblemReportID. Он объявлен обособленным (private), чтобы не допустить его изменения пользователями. Для однократной записи данных в этот объект введена функция SetProblemReportID. Поскольку указанный объект защищен от доступа со стороны подклассов, переопределенная функция SetProblemReportID не сможет изменить значение объекта ProblemReportID. Следовательно, исключается возможность несанкционированного изменения данных. Такой способ защиты данных существенно необходим, поскольку в дальнейшем будет показано, насколько важно обеспечить уникальность идентификатора сообщений для целостности базы данных.

Теперь специализируем базовый класс, добавив в него средства записи данных. Класс ModifiableProblemReport наследует от своего суперкласса операции-селекторы и поэтому может выполнить действия чтения-записи:

```
enum Boolean {False, True};

class ModifiableProblemReport : public ProblemReport {
public:

    ModifiableProblemReport ();
    ModifiableProblemReport (const ModifiableProblemReport&);
    virtual ~ModifiableProblemReport ();

    virtual int SetInternalPriority (Priority& APriority);
    virtual int SetStatusOfReport (Status& AStatus);
    virtual int SetReleaseOffFix (Release& ARelease);
    virtual int AddAuditor (Auditor& AnAuditor);
    virtual int RemoveAuditor (Auditor& AnAuditor);
    virtual int AddMaintenanceReport (MaintenanceReport& AMaintenanceReport);
    virtual int Update ();

    virtual Boolean IsUpdated () const;
};
```

Язык C++ не обеспечивает механизма обработки исключительных ситуаций, поэтому мы должны определить их самостоятельно. Операция-модификатор `SetReleaseOffFix` возвращает целое число. При правильном завершении операции возвращается ноль, если возникает ошибка при выполнении действий, возвращается код этой ошибки.

Отметим наличие операций введения и исключения аудиторов, позволяющих корректировать перечень сотрудников для уведомления с течением времени. Определена также операция добавления новых сообщений о ходе рассмотрения ошибки (но удаление таких сообщений запрещено, так как они должны храниться для анализа).

Теперь специализируем класс `ProblemReport` в подкласс `InitialProblemReport`, позволяющий осуществить регистрацию сообщения в базе данных:

```
class InitialProblemReport : public ProblemReport {
public:

    InitialProblemReport ();
    InitialProblemReport (const InitialProblemReport&);
    virtual ~InitialProblemReport ();

    virtual int SetSubmitter (Submitter& ASubmitter);
    virtual int SetSubmitterPriority (Priority& APriority);
    virtual int SetRelease (Release& ARelease);
    virtual int SetSummary (Summary& ASummary);
    virtual int SetDescription (Description& ADescription);
    virtual int SubmitProblemReport ();
    virtual Boolean IsSubmitted () const;
};
```

Аналогично определяются классы для следующих ключевых абстракций (текст не приводится):

- | | |
|------------|---|
| * Auditor | Подкласс от <code>Employee</code> — аудитор |
| * Auditors | Множество <code>Auditor</code> |
| * Creator | Подкласс от <code>Employee</code> — ответственный |

* Customer	Клиент
* Customers	Множество клиентов
* Description	Подробное описание проблемы
* Employee	Базовый класс — сотрудник
* Employees	Множество сотрудников
* Field-Representative	Подкласс от Employee — представитель по обслуживанию
* Group	Множество сотрудников
* Group Leader	Подкласс от Employee
* Group Member	Подкласс от Employee
* Maintenance-Report	Информация о причинах ошибки
* Maintenance-Reports	Множество сообщений
* Product	Продукт
* Products	Множество продуктов
* Registration	Регистрация пользователя продукта
* Registrations	Множество регистраций
* Release	Версия продукта
* Releases	Множество версий
* Submitter	Предъявитель сообщений
* Submitters	Множество предъявителей

Эти абстракции соответствуют словарю предметной области и позволяют пользоваться транзакциями, не вникая в реализацию на низком уровне (SQL-процедуры). Кроме того, наличие множества однотипных операций в процессе обслуживания транзакций (форматирование вывода на экран, обслуживание запросов и т.д.) позволяет создать множество общедоступных процедур.

Устранение семантических расхождений. Вернемся к вопросу о наличии программного интерфейса с абстракциями уровня базы данных. Имеется возможность (например, для реляционной СУБД Oracle) вызова из программы на C++ процедур создания курсора, операторов SQL, отмены запросов и т.д. [12]. Для программиста обеспечивается доступ ко всем ресурсам реляционной базы данных, включая создание таблиц, обзоров, выборок, модификации и удаления данных. Такие возможности представляют собой люк для доступа к базе данных и позволяют реализовать сколь угодно сложные прикладные программы.

Однако остается определенное расхождение между программным интерфейсом C++ и классами высокого уровня абстракции типа ProblemReport. Объект класса ModifiableProblemReport является элементарной абстракцией при выполнении транзакций, но его внутренняя реализация требует обращения к данным нескольких таблиц. Модификация сообщений может выполняться только отдельно для каждой таблицы, так как SQL не обладает свойствами модификаций соединений. Следовательно, необходимо осуществить связь между этими двумя уровнями абстракций, что является не совсем простой задачей.

Устранение семантических расхождений — одна из главных причин создания объектно-ориентированных баз данных. Для решения этой задачи рассмотрим подробнее два вида преобразования данных. Допустим, что нужно для конкретного продукта определить ответственную группу. Будем счи-

тать, что такая ответственная группа для каждого продукта только одна. Начнем определять классы Product и Group:

```
class Group {
public:
    virtual Name NameOfGroup () const;
};
class Product {
public:
    virtual Group GroupResponsibleForProduct () const;
protected:
    Group ResponsibleGroup;
};
```

Имея объект класса Product, мы можем записать следующий оператор:

```
TheName = TheProduct.GroupResponsibleForProduct().NameOfGroup()
```

При выполнении данного оператора возвращается наименование ответственной группы. Этот оператор не осуществляет операций над объектами, составляющими структуру класса. Конструктор объекта TheProduct не получает значений составляющих объектов. Идентификатор группы может быть получен независимо от других данных. При вызове оператора GroupResponsibleForProduct происходит следующее:

```
//      если объект Group уже считан из базы данных, то
//      возвращается объект Group
//      иначе
//      выполняется оператор SQL для выбора данной группы
//      создается объект Group
//      возвращается объект Group
```

В этом алгоритме доступ к базе данных осуществляется только при необходимости получения новой порции информации. В противном случае потребовалось бы большой объем памяти для дублирования данных по той же группе.

Теперь рассмотрим операцию Update, определенную в классе ModifiableProblemReport. Операция Update (изменение данных) является первичной транзакцией. Однако реализация этой операции не может быть простой и требует выполнения ряда функций с таблицами базы данных (с учетом существования и других процессов в системе). Кроме того, должна существовать возможность отмены запроса в случае каких-либо ошибок. То же можно сказать и о других видах преобразования данных. Операция Update выполняется следующим образом:

```
//      фиксируются (блокируются) все таблицы, входящие в соединение
//      для каждой из таблиц
//      модифицируются данные
//      таблицы деблокируются для последующего использования
```

Можно отметить, что приведенные примеры имеют много общего: работа ведется с несколькими таблицами, а внутри таблиц — со столбцами и строками. Поэтому целесообразно ввести следующие промежуточные классы:

- * Table (таблица)
- * Column (столбец)
- * Row (строка)

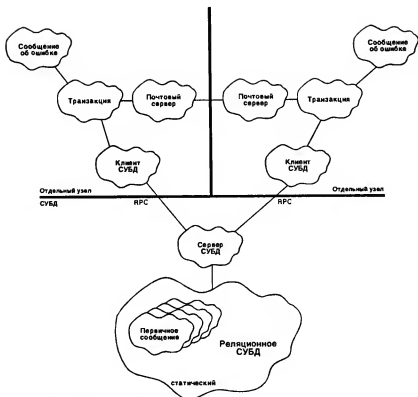


Рис. 10-9. Механизм передачи сообщений.

Эти классы являются аналогами соответствующих элементов SQL. В дальнейшем определяются специализированные подклассы:

* DeferredRecord (отложенная запись)

* DeferredField (отложенное поле)

которые реализуют отложенный доступ к данным. Это позволит создать классы Group и Product путем смешивания классов низкого уровня (используя множественное наследование).

Мы не будем более подробно останавливаться на описании таких промежуточных классов, поскольку они не вносят новых существенных элементов в процесс проектирования.

Механизм передачи сообщений

Из требований, предъявленных к системе, следует, что необходимо единственным способом реализовать две важные функции:

- * Уведомление руководителей и членов групп, ответственных за каждое сообщение.
- * Уведомление аудиторов (в частности, представителей по обслуживанию), отслеживающих ход рассмотрения ошибок.

Мы уже приняли решение по использованию электронной почты для передачи сообщений в сети. Это позволяет воспользоваться уже имеющимися средствами и интегрировать систему регистрации в хорошо известную инфраструктуру.

Использование средств электронной почты. Механизм передачи сообщений показан на рис. 10-9. Здесь, так же как в случае механизма SQL, узлы сети являются пользователями базы данных, которая реализована на отдельном компьютере-сервере. Каждый узел сети имеет локальный сервер электронной почты с программируемым интерфейсом для передачи сообщений удаленным пользователям. Сообщения читаются либо самими пользователями, либо фоновыми процедурами.

Рассмотрим теперь вытекающие из данного решения следствия. Анализ исходных требований приводит к следующему перечню операций с использованием данного механизма связи:

- * При предъявлении нового сообщения об ошибке, оно направляется в обслуживающий центр для предварительной классификации.
- * Если требуется предпринять дальнейшие действия (ошибка подтверждается), в обслуживающем центре определяются группы, ответственные за выявление причин.
- * Руководители групп периодически просматривают поступающие сообщения и передают их конкретным сотрудникам (членам этих групп).
- * По мере выяснения причин ошибок, члены групп формируют соответствующие сообщения или передают другим группам или программистам.
- * После окончательного выявления причин ошибки соответствующее сообщение «закрывается» с уведомлением клиентов через представителей по обслуживанию или маркетингу.
- * В любой момент времени заинтересованные лица (аудиторы) могут запросить информацию о состоянии зарегистрированных сообщений; любое изменение состояния дел доводится до всех аудиторов автоматически.

Программный интерфейс с электронной почтой может иметь следующий вид:

```
// <mailer.h>
```

```
typedef char*      UserName;
typedef UserName*  UserName[];

int CreateMailBox (UserName AUser);

int OpenMailBox ();
int CloseMailBox ();
int DisplayHeaders ();
int ReadMessage (int MessageNumber);
int Reply ();
int SendMessage (UserName      ToUser,
                  UserName      UserNamel,
                  char*          Subject,
                  char*          Text);

int NumberOfMessage ();
int NumberOfUnreadMessage ();
```

Этот интерфейс не удовлетворяет требованиям ООП, но можно найти компромиссное решение. Нам нужно создать смешанный класс, который соответствует объектам, связанным электронной почтой. Основу поведения этого класса составляет возможность пересылать объекты по электронной почте:



Рис. 10-10. Диаграмма классов сообщений.

```
// <mailer mixIn.h>

#include <mailer.h>

class MailerMixin {
public:

    int Notify (UserName Destination, UserNames CopiesTo);

protected:

    virtual char*      SubjectOfMessage () = 0;
    virtual char*      TextOfMessage () = 0;

};
```

SubjectOfMessage и TextOfMessage являются пустыми виртуальными функциями и должны быть определены в подклассах данного абстрактного класса. Функция Notify проста и типична для смешанных классов:

```
int MailerMixin::Notify (UserName Destination, UserNames CopiesTo) {
    return SendMessage (Destination, CopiesTo,
                        SubjectOfMessage (), TextOfMessage ());
}
```

Путем смешения нужных форм поведения мы можем сформировать требуемые нам классы. Приведем пример класса модифицируемого сообщения для пересылки по электронной почте:

```
class MailableModifiableProblemReport :      public ModifiableProblemReport,
                                              public MailerMixin {
protected:
    virtual char*      SubjectOfMessage ();
    virtual char*      TextOfMessage ();
}
```

Иерархия классов сообщений показана на рис. 10-10. Объекты приведенного выше класса объединяют свойства ModifiableProblemReport с возможностью уведомления аудиторов и других групп пользователей. Реализация этого класса завершается определением функций SubjectOfMessage и TextOfMessage, содержащих краткое изложение сообщения и его идентификатор.

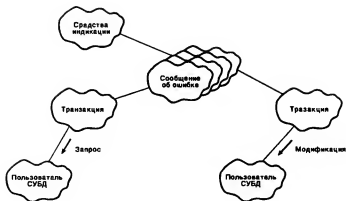


Рис. 10-11. Механизм формирования-отмены запросов.

Надежность сетевых коммуникаций. Из рис. 10-9 видно, что серверы электронной почты составляют связь между узлами сети. Допустим, что эти серверы реализуют процедуры обслуживания ошибочных ситуаций. Например, при нарушении связи с каким-либо узлом сети сервер почты попытается сам восстановить эту связь, либо передаст сообщение о невозможности установления связи с конкретным абонентом.

Это создает уверенность в том, что сообщения не могут просто исчезать, а возникающие сбои определенным образом обрабатываются. Сервер базы данных должен иметь гарантированную связь со своими пользователями. Необходимо, в частности, обеспечить совместное согласованное функционирование классов сервера и пользователя в случае нарушения связи между ними. Для обеспечения надежности придадим этим классам специальные свойства:

- Пользователи базы данных хранят очередь запросов на обслуживание. По мере реализации запросов очередь ликвидируется, но при нарушении связи запросы накапливаются.
- Сервер базы данных сообщает своим пользователям о принятии или отклонении запросов, чтобы они соответствующим образом отреагировали на эти факты.

Принятые нами проектные решения отвечают правилам ООП: по мере разработки проекта выполняется проверка логичности выбранной совокупности объектов и их связей. В результате формируются четко разграниченные ключевые абстракции и механизмы.

10.3. РАЗВИТИЕ ПРОЕКТА

Общие действия

Анализ вариантов. Чтобы выявить механизмы в системе регистрации ошибок, надо проследить за циклом прохождения какого-либо частного сообщения. На рис. 10-11 показаны основные объекты, взаимодействие которых определяет процесс функционирования системы.

Причиной активизации действий является уведомление разработчика о получении нового сообщения, которое поступает с помощью средств электронной почты. На основе полученной информации разработчик делает запросы в базу данных, в результате чего появляются экземпляры объектов класса `problemReport` (или его подклассов). В общем случае число объектов-сообщений может быть произвольным, например в случае запроса отчетов относительно конкретного продукта за последний месяц. Ответственный разработчик просматривает эти сообщения на терминале, а при необходимости делает твердую копию для последующего анализа; может быть также сформировано контрольное сообщение о результатах анализа сообщения.

Здесь возникает потенциальная возможность конфликта: пока разработчик А ведет анализ полученного сообщения, другой разработчик В может внести новые сообщения. В более общей формулировке необходимо создать механизм контроля и управления параллельными событиями, что характерно для любых параллельных систем. Аналогичная ситуация имеет место при бронировании мест на авиалиниях агентами, находящимися в различных городах.

Неверно было бы ставить вопрос о том, как устранить этот конфликт; следует поставить другой вопрос — как должна функционировать система.

Ответ на последний вопрос состоит в следующем: при попытках разработчика А внести изменения в состояние зарегистрированной проблемы, должно формироваться сообщение о запрете подобных действий, так как уже имеются более ранние версии анализа причин ошибки. После этого ответственному разработчику предстоит сделать выбор: отказаться от своего решения, поставить его в очередь, либо прямо связаться с разработчиком В. В любом случае управление передается от системы к пользователю.

Выбор решения. Теперь мы попробуем реализовать принятые принципиальные решения. Существует три возможных проектных варианта. В первом варианте базу данных можно сделать действительно распределенной, а не имитировать это качество на централизованной базе данных. В этом случае любые изменения со стороны пользователей вызывают широковещательные сообщения по всем возможным направлениям. Такой подход имеет существенное преимущество, связанное с почти полным отсутствием необходимости синхронизации сообщений. Однако дополнительная загрузка вычислителей и коммуникаций при этом заставляет отказаться от такого решения.

Во втором варианте на каждом сообщении можно ставить временную метку. Это не требует больших затрат вычислительной мощности и проработано теоретически, но практически неосуществимо. Чрезвычайно трудно в разнородной сети установить единое значение времени. Не помогает даже передача значения времени по сети, поскольку она требует прерывания сетевых процессов с определенной, достаточно большой частотой. Следовательно, этот вариант непригоден.

Третий вариант — самый простой. Каждое сообщение имеет свой особый идентификатор, сохраняющийся при всех изменениях. Копии сообщения дополняются специальным объектом `VersionNumber`, обозначающим версию сообщения. Вначале значение такого объекта равно нулю. При каждом обращении к базе данных сервер СУБД увеличивает номер версии, который впоследствии учитывается в процессах синхронизации.

Предположим, что разработчик А сделал запрос относительно сообщения Р впервые. Номер версии полученного сообщения будет 1. Обратившиеся по той же проблеме разработчики В и С получают сообщения с версиями 2 и 3. Предположим, что В попытается внести изменения в состояние проблемы.

Номер версии измененного сообщения (у разработчика В) станет равным 4. Попытки разработчиков А и С внести свои изменения в сообщение Р будут предотвращены, так как в базе данных уже имеется сообщение с более высокой версией.

Мы останавливаемся на этом подходе из-за его предельной простоты и универсальности. Для реализации изложенного принципа введем еще один смешиваемый класс для включения его в соответствующие классы-сообщения:

```
class VersionNumberMixin {
public:
    VersionNumberMixin ();
    VersionNumberMixin (const VersionNumberMixin&);
    virtual ~VersionNumberMixin ();

    int VersionNumber () const;

protected:
    int SetVersionNumber (int ANumber);
    void IncrementVersionNumber ();

private:
    int TheVersionNumber;
};
```

В этом классе отсутствуют виртуальные функции, поскольку нет необходимости переопределять функции в подклассах.

Объект TheVersionNumber инициализируется конструктором с нулевым значением, а селектор позволяет читать его значение:

```
int VersionNumberMixin::VersionNumber () {
    return TheVersionNumber;
};
```

Функция IncrementVersionNumber увеличивается на 1 номер версии:

```
void VersionNumberMixin::IncrementVersionNumber () {
    TheVersionNumber++;
};
```

Функция SetVersionNumber позволяет установить номер версии только, если заданный номер больше текущего:

```
int VersionNumberMixin::SetVersionNumber (int ANumber) {
    if (ANumber > TheVersionNumber)
        return TheVersionNumber = ANumber;
    else
        return 0;
}
```

Действия пользователя. Выше мы уже определили семь функций пользователей. Все они различны, поэтому создадим иерархию класса сотрудников с подклассами предъявителя, аудитора, руководителя группы и т.д.

Предполагается наличие соответствующих классов транзакций, предназначенных для смешения. Например, предъявитель, аудитор, представитель по обслуживанию и представитель по маркетингу имеют возможность инициировать сообщения, но не могут их модифицировать. Напротив, руководители групп и ответственные разработчики кроме запросов к базе данных могут вносить в них изменения.

Где должны выполняться такие действия, как модификация сообщений? Эти действия могут инициироваться пользователем, но не выполняться им. При этом должны быть обеспечены надежность и безопасность системы путем предоставления каждому пользователю строго ограниченных и контролируемых ресурсов.

Это достигается путем смешения классов, вносящих свои аспекты поведения, например, следующим образом:

```
class Update {
public:

    Updater (ModifiableProblemReport& AReport);
    Updater (const Updater&);
    virtual ~Updater ();

    virtual int          Commit(TheReport );
    ModifiableProblemReport ReportToBeUpdated () const;

private:
    ModifiableProblemReport TheReport;
};
```

Для пользователей, которым разрешено изменение информации, определяется следующий класс:

```
class EmployeeWithUpdateRights : public Employee, public Updater {
public:
    EmployeeWithUpdateRights (ModifiableProblemReport& AReport);
};
```

Такие же классы определяются для других видов функций пользователей.

Утилиты и генераторы версий

Выделение утилит. Стиль проектирования, основанный на использовании стандартных механизмов, не является догмой. Мы могли бы создать специальную базу данных или реализовать собственный «доморощенный» протокол связи, чтобы обеспечить защиту данных и уведомление о сообщениях. Однако мы предпочитаем, где возможно, использовать стандартные средства. Это оправдывается, особенно в проектах, имеющих последовательно-итеративный характер, поскольку гарантируется надежность используемого кода и более просто осуществляется адаптация к требованиям задачи.

Уже говорилось, что проектирование базы данных представляет собой последовательный-итеративный процесс, соответствующий методологии ООП. Единственный способ убедиться в верности каждого принимаемого проектного решения — пробная эксплуатация созданной системы. Комбинация анализа, тестирования и личного опыта приближает нас к принятию верных решений, но практика требует «прощупать» первые версии базы данных, чтобы

адаптировать ее к конкретным условиям повседневной эксплуатации. Хавришкевич [13] утверждает, что все основные неувязки в базах данных, которые можно выявить в процессе проектирования и эксплуатации, требуют лишь небольших усилий для устранения и значительного улучшения характеристик. Приведем их перечень:

- * «Слишком много логических записей участвует в осуществлении запроса.
- * Слишком много шагов доступа к данным требуется для удовлетворения запроса.
- * Слишком много вспомогательных файлов и операций сортировки.
- * Слишком много физических записей требуется для удовлетворения запроса.
- * Чрезмерен объем требуемой памяти.
- * Перегрузка функциями СУБД или ОС».

Используя стандартные средства, мы можем начать реализацию и настройку на более ранних стадиях проекта. Кроме уже рассмотренных механизмов, мы должны создать программный интерфейс, связывающий все ключевые элементы системы. С учетом уже определившейся инфраструктуры в поле нашего зрения попадают две группы специальных средств: утилиты и генераторы версий.

Под утилитами понимаются наборы сложных операций, относящихся к одному или нескольким классам. В гл. 3 было показано, как происходит увеличение числа утилит класса по мере уточнения программистом особенностей работы программы. Чтобы не увеличивать число процедур-утилит в системе, целесообразно собрать их в одном месте и сделать общедоступными.

О некоторых утилитах мы уже говорили, но анализ предметной области позволяет выделить еще несколько кандидатур на роль утилит:

- | | |
|-----------------------|---|
| * Утилиты запросов | Выражения, позволяющие программисту реализовывать запросы высокого уровня (не обращаясь к SQL). |
| * Утилиты почты | Общепользовательские средства связи по электронной почте, включая передачу сообщения группе пользователей по списку, или особые формы форматирования сообщений. |
| * Утилиты регистрации | Средства регистрации информации о работе всей системы в целом (сбор сетевых средств, открытие и закрытие базы данных и т.п.). |

Рассмотрим подробнее последнюю группу утилит, используемых для регистрации различных видов событий, происходящих в системе. Можно выделить следующие виды событий:

- * Отладочные сообщения.
- * Сообщения о выполнении запросов.
- * Сообщения об ошибках.
- * Предупреждающие сообщения.
- * Сообщения о сбоях в системе.
- * Справочные уведомления.

Каждому из таких видов событий может соответствовать свой класс. Но благодаря некоторой общности целесообразно реализовать их в виде подклассов суперкласса Message:

```
class Message {
public:
    Message (char* TheString);
    Message (const Message&);
    virtual ~Message();
    virtual char* Value ();
private:
    char* TheValue;
};
```

Формы представления различных видов сообщений также отличаются,

например:

* Отладка	89/08/25 09:41:06	???	Создано временное окно
* Выполнено	89/08/25 09:43:06	+++	Копирование завершено
* Ошибка	89/08/25 09:41:06	---	Невозможно отправить сообщение
* Предупреждение	89/08/25 10:00:76	!!!	Недостаточно места
* Сбой	89/08/25 11:41:06	***	Пароль неверен
* Уведомление	89/08/25 23:16:57	...	Ожидание ввода пароля

Общий для всех сообщений является наличие даты и времени их возникновения. Различия состоят в специальных символах перед сообщением.

Теперь определим класс Log, который лишь заносит сообщения в файл:

```
class Log {
public:
    Log ();
    Log (const Log&);
    virtual ~Log ();

    virtual int      Open (char* AFile);
    virtual int      Close ();
    virtual void      Put (Message& TheMessage);

private:
    ofstream TheFile;
};
```

Можно добавить утилиту Filter, которая позволяет читать файлы регистрации событий и делать из них соответствующие выборки:

```
int Filter (File *LogFile,
            File *Destination,
            Boolean IncludeDebugMessages,
            Boolean IncludeSuccessMessages,
            Boolean IncludeFailureMessages,
            Boolean IncludeWarningMessages,
            Boolean IncludeErrorMessages,
            Boolean IncludeNotes);
```

Используя эту утилиту, можно строить более сложные вспомогательные средства. Ниже мы рассмотрим такие вспомогательные средства, особенно полезные для утилит регистрации.

Назначение генераторов версий. Система регистрации ошибок может содержать большое число форм и видов сообщений. Реализация в большой системе этих видов и форм не представляет труда, но чрезвычайно утомительно, поэтому в коммерческих приложениях популярны генераторы версий (4GL для языков четвертого поколения).

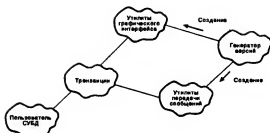


Рис. 10-12. Утилиты транзакций.

Из рис. 10-12 видно, что все транзакции, связанные с пользователями, применяют утилиты сообщений и графического интерфейса. Они могут автоматически создаваться специальными генераторами, которые на основе спецификаций сообщений и видов индикации формируют нужные шаблоны. Задача проектировщика, таким образом, сводится не к программированию сотен видов графических сообщений, а к созданию одного-двух генераторов для формирования таких утилит. Подобный подход на практике существенно сокращает объем программного кода для хорошо формализованных приложений.

Проектирование генератора версий представляет самостоятельный интерес и здесь не рассматривается. Программисты могут воспользоваться коммерческими версиями таких генераторов при необходимости. Написать собственный проблемно-ориентированный генератор с использованием методологии ООП.

Модульная архитектура

В процессе развития проекта мы создали ряд классов и утилит, составляющих несколько файлов. Опыт подсказывает, что система регистрации ошибок в полном объеме составит 20—30 тысяч строк кода на языке C++. Отказ от языка C++ и выявления общности между ключевыми абстракциями (не объектно-ориентированная версия системы) приведет к увеличению размера кода примерно на 50%.

Поскольку мы создаем систему, а не отдельную программу, нам необходимо как можно раньше установить архитектуру модулей. Соответствующее проектное решение отражено на рис. 10-13. Как и следовало ожидать, система глубоко структурирована. Классы и объекты нижнего уровня отнесены к сетевой подсистеме, в которой реализуется механизм связи. Подсистема СУБД содержит средства SQL с классами высокого уровня, а подсистема транзакций различные утилиты и генераторы версий. Самый верхний пользовательский уровень представляет подсистема User Application (пользовательские приложения).

Предложенная структура не только содержит форму физической организации модулей в систему, но и позволяет распределять ресурсы группы программистов. Впоследствии она станет основой организации эксплуатации версий системы. При распределении модулей учитывалось и то, насколько часто в той или другой подсистеме возможны изменения.

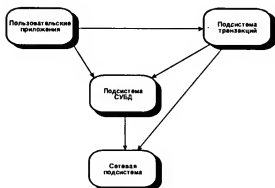


Рис. 10-13. Диаграмма модулей системы регистрации.

10.4. МОДИФИКАЦИЯ

Расширение функциональных свойств

После получения действующей версии системы потребуется некоторое время для работы с ней администратора базы данных. Если наша система заменила ранее существовавшую, администратор может реализовать переиос уже существующих данных в новую систему. Этому поможет то, что мы предусмотрели для этого в системе ряд инструментальных вспомогательных средств. Могут выявиться ошибки нашего анализа. Причина этого не обязательно состоит в нашей некомпетении, а может объясняться постоянным совершенствованием всех видов деятельности, в том числе и бизнеса.

Рассмотрим возможность внесения в систему какого-либо изменения. Допустим, что предъявитель сообщения ввел данные об ошибке, относящейся к нескольким версиям различных программных продуктов. Возможно, что два сообщения будут содержать сведения об одной и той же ошибке, а нам не хотелось бы дважды тратить усилия на анализ. Возникает потребность в механизме, позволяющем расщеплять или группировать сообщения. Для этого требуется создать смешиваемый класс *Genealogy*, позволяющий реализовать между сообщениями отношения предок-наследник. Если сообщение *P* расщепляется на *X* и *Y*, то *P* является предком для *X* и *Y*, а они в свою очередь наследниками *P*. То же относится и к процессу группировки.

Учитывая это, внесем изменения в реализацию транзакций подклассов класса *ProblemReport*. При этом изменение расщепленного сообщения сводится просто к изменению его наследников. Для обеспечения координации изменения одного из наследников сообщают аудиторам, связанным со всеми исходными сообщениями. Внесение в систему существенных функциональных изменений укладывается в рамки принятых проектных решений.

Изменение исходных требований

Предположим, что мы решили дополнить систему мощным графическим интерфейсом, позволяющим легко работать служащим без специальной подготовки. Оказывается, что основа системы при этом сохранится. Все механизмы, кроме генератора версий, остаются без изменений. Независимость механизмов, основанная на ожидании изменений лишь в некоторых из них, позволяет реализовать сколь угодно развитый графический интерфейс пользователя. В отличие от интерфейса, рассмотренного в предыдущей главе, в системе регистрации ошибок требуется другой многооконный интерфейс, создание которого мы здесь не рассматриваем.

Дополнительная литература

Особенности реляционных баз данных подробно описаны в работах Date [E, 1981, 1983, 1986]. Описание стандарта SQL приведено в работе Date [E, 1987]. Различные подходы к анализу данных даны в работах Weryard [B, 1984], Nawryszkiewicz [E, 1984] и Ross [F, 1987].

Интеграция традиционных СУБД с объектным подходом составляет сущность объектно-ориентированных баз данных. Исследования в этом направлении приведены в работах Davis [H, 1983], Kim, Lochovsky, [E, 1989], Zdonik, Maier, [E, 1990].

Краткое изложение языка C++ с примерами дано в приложении.

Глава 11

Common Lisp Object System. Система дешифрования

Живые организмы способны проявлять очень сложные формы поведения на основе мало еще изученных механизмов мышления. Вспомните, как вам приходилось планировать маршруты поездок по городу, чтобы выполнить массу поручений. Или как в плохо освещенном помещении удастся определить расположение предметов, чтобы избежать столкновения. Или еще о том, как вам удавалось вести беседу с человеком в присутствии множества одновременно говорящих людей. Ни одну из таких задач невозможно решить на основе строгого алгоритма. Планирование маршрута является многомерной задачей с вариантами. Передвижение в темноте требует принятия решений на основе неполных и размытых (буквально) визуальных данных. Выделение речи из множества разговоров требует умения находить полезную информацию в условиях шума. Эти и подобные им проблемы решаются исследователями в области искусственного интеллекта (ИИ) с целью моделирования процессов познания. Создаются интеллектуальные системы, которые имитируют некоторые стороны поведения человека.

Erman, Lark, Hayes-Roth установили, что «интеллектуальные системы отличаются от традиционных рядом признаков, некоторые из которых не являются обязательными:

- * Они способны достигать целей, изменяющихся во времени.
- * Они способны сопоставлять, использовать и преобразовывать знания.
- * Они справляются с многообразием специальных подсистем, варьируя методы.
- * Они обеспечивают интеллектуальный интерфейс с пользователем и другими системами.
- * Они сами планируют свои ресурсы и концентрируют их в нужном направлении» [2].

Реализация в системе любого из этих требований является непростой задачей. Еще сложнее сделать интеллектуальную систему для использования в различных прикладных областях (например, для медицинской диагностики и диспетчеризации авиарейсов): системы искусственного интеллекта (ИИ) редко создаются для всеобщего использования. Несмотря на значительные преувеличения успехов энтузиастов ИИ, работы в этой области дали немало хороших практических идей, в частности относительно представления знаний, методов решения задач на основе экспертных систем и методологий «классной доски». В данной главе рассматриваются подходы к созданию интеллектуальной системы для расшифровки криптограмм на основе метода «классной доски», в значительной степени аналогичного способу расшифровки, применяемому человеком. Мы будем иметь повод убедиться, что методы OOD хорошо соответствуют этой задаче¹⁾.

¹⁾ Для решения задачи будем использовать язык CLOS, имея в виду, что CLOS ориентирован не столько на задачи ИИ, сколько на задачи связанные, с последовательно-итеративными процессами, что характерно для большинства сложных программных систем.

Задача шифрования

Шифрование «охватывает методы, позволяющие сделать данные непонятными для посторонних» [1]. Алгоритмы шифрования преобразуют сообщения (исходный текст) в зашифрованный текст (криптограмму) и наоборот.

Одним из наиболее известных еще со времен Древнего Рима видов шифрования является алгоритм подстановки (замены знаков). При этом каждая литера в алфавите исходного текста заменяется другой литерой. Например, можно сдвинуть все буквы алфавита: А заменяется на В, В — на С, а Z — на А. Тогда следующий исходный текст:

CLOS is an object-oriented programming language

превращается в криптограмму:

DMPT jt bo pckfdu-psjfoufe qsphsbnnjoh mbohvbhf

Чаще всего замена делается случайным образом, А заменяется на G, В — на J и т. д. Возьмем для примера такую криптограмму:

POG TBCER CQ TCK AL S NGELCH QZBBR SBAJ G

Для подсказки уточним, что буква С соответствует букве О исходного текста. Предположим, что для шифрования текста использован алгоритм подстановки, что существенно упрощает задачу, в общем случае процесс дешифровки не будет столь тривиальным. В процессе расшифровки приходится использовать и метод проб и ошибок, когда мы делаем пробное предположение о варианте замены знаков и рассматриваем последствия этого предположения. Можно, например, начать расшифровку с предположения о том, что одно- и двухбуквенные слова в криптограмме соответствуют наиболее употребительным словам языка (I, a, or, it, in, of, on). Подставляя эти буквы в другие слова, мы можем увидеть вероятные значения других литер. Например, если трехбуквенное слово начинается с литеры «о», то это могут быть слова one, our, off. Знание фонетики и грамматики также может способствовать дешифровке. Например, следование подряд двух одинаковых литер с малой вероятностью может означать «qq». Наличие в окончании слова буквы «g» позволяет сделать предположение о наличии суффикса «ing». На еще более высоком уровне абстракции логично предположить, что группа слов «it is» более вероятна, чем «if is». Необходимо учитывать и структуру предложений, которая включает обычно существительные и глаголы. Если выясняется, что в предложении есть глагол, но нет существительного с ним связанного, то нужно отвергнуть сделанные ранее предположения и начать новый поиск вариантов. Иногда приходится возвращаться в обратном направлении, если сделанное предположение вступает в противоречие с другими предположениями. Наша задача: найти систему, которая преобразует криптограмму в исходный текст, исходя из предположения о наличии простой подстановки литер.

11.1. АНАЛИЗ

Определение границ предметной области

Выделенный в рамку текст даст описание рассматриваемой предметной области, а именно дешифровки криптограмм. В общем случае процесс дешифровки является крайне сложным и неподатливым даже для очень «хитрых» методов. Существует, например, стандарт шифрования DES (используемый для шифровки в самых разных областях применения) с глубокой математической основой, устойчивый ко всем известным методам дешифровки. Но наша задача достаточно проста, поскольку мы ограничимся методами простой замены букв.

Попробуем решить следующую криптограмму и отметить, как мы это делаем (не забегая вперед):

Q AZWS DSSC KAS DXZNN DASNN

Для подсказки отметим, что буква W соответствует букве V исходного текста. Попытка перебрать все возможные варианты совершенно бессмысленна. Предполагая, что алфавит содержит только 26 прописных английских букв, получим $26!$ (около 4.03×10^{26}) возможных комбинаций. Следовательно, нужно искать обходной метод решения задачи например, использовать знания о структуре слов и предложений и делать правдоподобные допущения. Как только мы исчерпаем явные решения, мы будем делать вероятные предположения и продвигаться дальше. Если обнаружим, что предположение приводит к тупику или противоречию, то вернемся назад и сделаем другую попытку.

Будем отмечать шаги нашего поиска:

1. Используя подсказку, заменим W на V.

Q AZVS DSSC KAS DXZNN DASNN

2. Первое слово из одной буквы, вероятно, A или I, предположим, что это A.

A AZVS DSSC KAS DXZNN DASNN

3. В третьем слове должны быть гласные звуки и вероятно, что это двойные буквы. Это не может быть UU или II, а также AA (буква A уже использована). Попробуем вариант EE.

A AZVE DEEC KA E DXZNN DAENN

4. Четвертое слово состоит из трех букв и оканчивается на E, это очень похоже на THE.

A HZVE DEEC THE DXINN DHENN

5. Во втором слове нужна гласная и для этого подходят I, O, U. Только I дает значение, подходящее по смыслу.

A HIVE DEEC THE DXINN DHENN

6. Можно найти несколько слов с двойной буквой E (DEER, BEER, SEEN). Грамматика требует, чтобы третье слово было глаголом, поэтому остановимся на SEEN.

A HIVE SEEN THE SXINN SHENN

7. Смысл в полученном предложении отсутствует, поскольку крапивница (HIVE) не может видеть (SEEN), значит, первоначальные предположения содержат ошибку. Похоже, что выбор гласной буквы был неверен, и приходится вернуться назад.

A HAVE SEEN THE SXINN SHENN

8. Посмотрим на последнее слово, двойная буква S в конце не дает осмысленного значения и уже использована ранее, а вот LL может иметь место.

A HAVE SEEN THE SXINN SHELL

9. Последнее слово является частью составного существительного. В качестве второй его части по шаблону S? ALL подходит слово SMALL.

A HAVE SEEN THE SMALL SHELL

Таким образом, решение найдено. Из процесса решения можно сделать три заключения:

- * Применены знания о грамматике, составе слов и чередовании согласных и гласных.
- * Сделанные предположения направлялись на ключевое место, а на основе знаний строилась цепочка дальнейших рассуждений.
- * Рассуждения во времени менялись. Иногда делались выводы от общего к частному (слово из трех букв с окончанием на E — это THE), а иногда от частного к общему (?EE? может соответствовать DEEF, BEER, SEEN, но глагол только SEEN).

Изложенный подход известен как метод «классной доски». Этот метод впервые предложен Ньюэллом (Newell) в 1962 году, а позднее использован Рэдди и Эрманом (Reddy, Erman) в проектах Hearsay, Hearsay II по распознаванию речи [3]. Эффективность этого метода подтвердилась, и метод был использован в других областях (выделение сигналов, моделирование молекулярных объемных структур, распознавание образов и планирование [4]). Этот метод показал хорошие результаты для представления декларативных знаний и пространственно-временную эффективность в сравнении с другими подходами [5].

Архитектура, основанная на «классной доске»

Ингломор и Морган (Englemore, Morgan) для пояснения метода «классной доски» использовали аналогию составления группой людей изображения из фрагментов-вырезок:

«Вообразим себе комнату с большой классной доской, рядом с которой находится группа людей, держащих в руках фрагменты большого изображения. Процесс начинают добровольцы, которые размещают на доске наиболее характерные фрагменты изображе-

ния (предположим, что они приклеиваются к доске). Далее каждый участник группы смотрит на оставшиеся у него фрагменты и определяет наличие таких, которые подходят к находящимся на доске. Участник, нашедший такое соответствие, подходит к доске и производит соответствующее изменение. В результате фрагмент за фрагментом занимают нужное место по мере подхода к доске других участников этого процесса. При этом не существенно, что один из участников имеет больше фрагментов, чем другой. Все изображение будет полностью собрано без всякого обмена информацией между членами группы. Каждый участник активизируется самостоятельно и знает, когда ему нужно включиться в процесс. Никакого порядка подхода к доске заранее не устанавливается. Совместное поведение регулируется только состоянием информации на классной доске. Наблюдение за этим процессом демонстрирует его ступенчатый характер (по одному фрагменту за подход) и то, что оно основано на согласии (согласие на установку очередного фрагмента). Это существенно отличается от попытки начать с одного из углов и последовательной проверки каждого фрагмента на возможность помещения в соседнее поле» [6].

Из рис. 11-1 видно, что основу метода составляют три элемента: классная доска, совокупность источников знаний и управляющий этими источниками контроллер [7]. Отметим, что последующее определение прямо соответствует принципам объектного подхода. Нии (Nii) установил: «Классная доска нужна для того, чтобы хранить данные о ходе и состоянии решаемой задачи, которые (данные) используются и формируются источниками знаний. Классная доска объединяет объекты из пространства решений. Эти объекты иерархически группируются по уровням анализа и вместе со своими атрибутами образуют словарь пространства решений» [8].

Инглмер и Морган уточняют, что «необходимые для решения задачи знания о предметной области разделены по нескольким независимым источникам. Каждый источник знаний имеет своей целью сделать информационный вклад, который приблизит решение задачи. Текущая информация из каждого источника помещается на «классной доске» и модифицируется в соответствии с содержанием знаний. Формой представления источников знаний являются процедуры, наборы правил или логические заключения» [9].

Источники знаний (обозначенные KS) являются зависимыми от предметной области. В системах распознавания речи могут быть сведения о фонемах, словах и предложениях. В системах распознавания образов — сведения о элементарных структурах изображения, таких, как стыки линий или участки одинаковой плотности, а на более высоком уровне абстракции об объектах, относящихся к конкретной задаче (дома, дороги, поля, автомобили и люди).

В общем случае источники знаний соответствуют иерархической структуре объектов, размещаемых на «классной доске». Более того, каждый источник оперирует с объектами строго определенных уровней иерархии в качестве источников информации и в качестве получателей сообщений. Например, в системе распознавания речи источник знаний о словах должен наблюдать за потоком фонем (низкий уровень абстракции), чтобы сформировать данные о выделенном слове (более высокий уровень абстракции); путем фильтрации списка возможных слов (снова более низкий уровень абстракции) этот источник может проверять гипотезы.

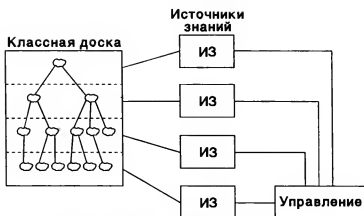


Рис. 11-1. Основные элементы «классной доски».

Два указанных приема поиска решения называются соответственно прямой и обратной цепью суждений. Прямая цепь суждений позволяет перейти от более частных предположений к более общим, а обратная цепь, начинаясь от некоторой гипотезы, позволяет проверить эту гипотезу на известных предположениях. Вот почему управление на «классной доске» является согласительным, в зависимости от обстоятельств источники знаний могут активизировать либо прямые, либо обратные цепи суждений. Источники знаний, как правило, состоят из двух компонент-предусловия и действия.

Предусловие — это состояние «классной доски», которое представляет «интерес» для данного источника знаний (способно его активизировать). Например, в распознавании образов предусловием может быть наличие прямой области изображения (которая может означать дорогу). Выполнение предусловий заставляет источник знаний сфокусировать внимание на конкретном участке «классной доски», а затем привести в действие соответствующие процедурные правила или знания.

В этих условиях очередность активизации является излишней: если источник знаний обнаруживает полезные данные для решения задачи, он дает сигнал об этом контроллеру «классной доски». Это напоминает жест рукой, указывающий на желание сделать сообщение. Из нескольких источников, делающих такой жест, контроллер выбирает наиболее перспективный с его точки зрения и передает ему управление (разрешение на выполнение действий).

Анализ источников знаний

Вернемся теперь к поставленной задаче и подумаем, какие источники знаний будут полезны для ее решения. Наиболее часто инженеры знаний в такой ситуации просто садятся рядом с экспертом по предметной области и начинают выявлять приемы (эвристики), применяемые экспертом при решении аналогичных задач. В нашем случае для этого придется расшифровать некоторое количество криптограмм и отметить особенности процесса поиска решения.

В результате выявляется тринадцать источников знаний, относящихся к проблеме, которые приведены в списке:

* Общие приставки	Наиболее частое начало слов (например, ge, anti, un).
* Общие суффиксы	Наиболее частое окончание слов (ly, ing, es, ed).
* Согласные буквы	Буквы не являющиеся гласными.
* Прямая подстановка	Подсказки, как часть выражения.
* Сдвоенные буквы	Наиболее часто сдвигаемые буквы (tt, ll, ss).
* Частота букв	Вероятность появления букв в тексте.
* Правильные строки	Допустимые и недопустимые сочетания букв (например, qu и zg).
* Проверка по шаблону	Слова соответствующие шаблону.
* Структура предложения	Грамматика, включая значения существительных и глаголов.
* Короткие слова	Варианты одно-, двух-, трех- и четырехбуквенных слов.
* Решение	Найдено ли решение или имеет место тупик.
* Гласные буквы	Буквы не являющиеся согласными.
* Структура слова	Расположение гласных и общая структура существительных, глаголов, прилагательных, наречий, приставок, союзов и т.д.

Исходя из объектно-ориентированного подхода, все эти источники являются потенциальными классами, объекты которых имеют состояние (знания), характеризуются поведением (знание суффикса может влиять на значение слова) и обладают индивидуальностью (знания о коротких словах независимы от проверки по шаблону).

Можно установить иерархию указаний источников знаний. В частности, существует группы источников знаний о предложениях, о словах, о группах букв и об отдельных буквах. Такая иерархия соответствует объектам, выносимым на «классную доску»: предложения, слова, строки и буквы.

11.2. ПРОЕКТИРОВАНИЕ

Проектирование «классной доски»

Теперь у нас есть все, чтобы приступить к решению поставленной задачи с использованием метода «классной доски». Это классический пример повторного использования метода решения на уровне проекта. Метод «классной доски» предполагает, что среди объектов верхнего уровня системы имеют место: классная доска, несколько источников знаний и контроллер. Остается только определить классы и объекты предметной области, которые специализируют эти, наиболее общие, абстракции.

Объекты классной доски. Объекты образующие содержание классной доски, отвечают трем уровням абстракции знаний и образуют три следующих класса:

* sentence	Полная криптограмма
* word	Отдельное слово в криптограмме

- * cipher-letter Отдельная буква в слове

Источники знаний должны пользоваться общей информацией о сделанных в процессе решения предположениях, поэтому в число объектов классной доски включается следующий класс:

- * assumption Сделанные предположения (допущения)

Необходимо также знать полиный буквенный алфавит, используемый в тексте и криптограмме, что дает последний класс:

- * alphabet Алфавит текста, алфавит криптограммы и их соотношение

Есть ли между этими пятью классами что-то общее? Ответ утвердительный: все они соответствуют объектам классной доски и существенно отличны от других объектов (например, от контроллера). Поэтому вводится следующий суперкласс для всех этих объектов:

```
(defclass blackboard-object ())
```

С точки зрения внешнего поведения определим для этого класса две операции:

- * add-object Добавить объект на классной доске
- * remove-object Удалить объект с доски

Зависимости и документация. Предложения, слова и буквы также связаны определенной общностью: все они зависят от конкретного источника знаний и должны уметь обращаться к этим источникам. Источники знаний в свою очередь должны получать информацию об изменениях соответствующих объектов. Это напоминает механизм зависимостей языка Smalltalk, использованный в гл. 9. Итак, запишем:

```
(defclass dependent ()
  ((the-reference :accessor reference)))
```

Для этого класса (dependent) определен всего один слот the-reference, обозначающий список источников знаний, связанных с данным объектом. К числу операций данного класса относятся:

- * add dependency Добавляет ссылку на источник знаний
- * remove-dependency Удаляет ссылку на источник знаний
- * dependency-p Селектор: определяет зависимость объекта от заданного источника знаний
- * each-dependency Итератор: проверка всех зависимостей

Использован стиль именования предикатов языка CLOS (например, dependency-p) за счет суффикса -p. Зависимость является особым свойством, которое может примешиваться к другим классам. Например, буква является объектом классной доски и зависимостью, что позволяет смешать два этих класса для получения нужного поведения. Механизм смешения увеличивает

возможность повторного использования классов и сохраняет их независимость в процессе проектирования.

Буквы имеют общие свойства с алфавитом: объекты этих классов могут иметь взаимные отношения и допущения. Так, некоторый источник знаний может допустить, что буква «к» в шифре соответствует букве «р» исходного текста. Поэтому введен еще один класс-смесь:

```
(defclass assumable-object ())
```

В нашей системе предположения имеют место только в отношении отдельных букв, но не о словах и предложениях. Можно, например, предположить, что какая-либо буква шифра соответствует некоторой букве алфавита. Но алфавит состоит из множества букв, каждая из которых может быть поставлена в однозначное соответствие с шифром. Это отличие букв шифра от букв алфавита объясняет отсутствие слотов в классе `assumable-object`; слоты появятся при определении производных классов. Следовательно, класс `assumable-object` служит исключительно для определения группы общих операций, которые будут реализованы в производных подклассах. Определим следующий набор операций для экземпляров этого класса:

- | | |
|----------------------------|--|
| * state assumption | Сделать допущение |
| * retract-assumption | Отменить допущение |
| * plain-assoc | Вернуть шифрованный эквивалент для заданной буквы текста |
| * cipher-assoc | Вернуть текстовый эквивалент для заданной буквы шифра |
| * plain-letter-defined-p | Селектор: определена ли заданная буква текста? |
| * cipher-letter-defined-p | Селектор: определена ли заданная буква шифра? |
| * plain-letter-asserted-p | Селектор: найдено ли значение для данной буквы текста? |
| * cipher-letter-asserted-p | Селектор: найдено ли значение для данной буквы шифра? |

Из приведенного описания видно, что установлено два вида соотношений шифр-текст: одно — временное определение и другое — окончательно доказанное значение. По мере расшифровки криптограммы может делаться множество различных предположений о соотношении букв шифра и текста, но в конце концов находятся окончательные значения такого соотношения для всего алфавита. Этот подход находит явное выражение в описании классов. Сначала определяем класс `assumption`:

```
(defclass assumption (blackboard-object)
  ((the-knowledge-source
    :accessor the-knowledge-source
    :initarg :the-knowledge-source)
   (the-reason
    :accessor the-reason
    :initarg :the-reason)
   (the-plain-letter
    :accessor the-plain-letter
    :initarg :the-plain-letter)
   (the-cipher-letter
    :accessor the-cipher-letter
    :initarg :the-cipher-letter)
   (the-assumable-objects
    :accessor the-assumable-objects)))
```

Этот класс является объектом «классной доски», поскольку информация о сделанных предположениях используется всеми источниками знаний. Отдельные слоты означают следующие свойства:

- * the-knowledge-source Источник знаний, из которого исходит сделанное предположение
- * the-reason Основополагающее для сделанного предположения
- * the-plain-letter Буква исходного текста, для которой сделано предположение
- * the-cipher-letter Предполагаемое значение шифра для буквы исходного текста
- * the-assumable-objects Объекты «классной доски», которые затрагивают сделанное предположение

Необходимость каждого из приведенных слотов в значительной степени объясняется природой предположений: какой-либо источник знаний формирует предполагаемое соотношение буква-шифр на основании определенных причин (обычно путем реализации некоторого правила). Назначение последнего слота (the-assumable-object) менее очевидно. Его необходимость вызвана возможностью возникновения обратных цепочек суждений. Если сделанное предположение не подтвердится, то нужно соответственно изменить состояние объектов, которые воспользовались сделанным ранее предположением (через механизм зависимости). Далее определим подкласс assertion:

```
(defclass assertion (assumption)
  ())
```

Общим для классов assumption и assertion являются следующие операции:

- * state-assumption Сделать допущение
- * retract-assumption Отвергнуть допущение
- * retractable-p Селектор: является ли допущение временным?

Для всех сделанных предположений (допущений) значение предиката retractable является истинным, а для доказанных окончательно ложным. Доказанное предположение уже нельзя изменить или отвергнуть.

Проектирование объектов «классной доски». Необходимо теперь ввести в проект классы для предложения (sentence), слова (word), буквы шифра (cipher-letter) и алфавита (alphabet). Предложение — это объект «классной доски», содержащий список слов, составляющих данную фразу. Исходя из этого, запишем:

```
(defclass sentence (blackboard-object dependent)
  ((the-words :accessor the-words)))
```

В дополнение к операциям add-object и remove-object (определенным в суперклассе blackboard-object) и четырем операциям, унаследованным от класса dependent, добавлено еще две следующие:

- * sentence-value Текущее значение предложения

- * solved-p Имеет значение true (истина), если все слова в предложении полностью расшифрованы

Первоначально sentence-value совпадает с текстом криптограммы. А когда значение solved-p станет истинным, то с помощью операции sentence-value можно прочесть исходный (расшифрованный) текст. Слово также является объектом «классной доски», включая свойства зависимости, и состоит из последовательности букв. Для удобства оперирования источниками знаний в класс введены указатели от слова на соответствующее предложение, а также на предыдущее и последующее слова в данном предложении. Описание класса word выглядит так:

```
(defclass word (blackboard-object dependent)
  ((the-letter      :accessor the-letters)
   (the-sentence    :accessor the-sentence)
   (the-previous-word :initarg :the-sentence)
   (the-next-word   :accessor the-previous-word)
   (the-next-word   :accessor the-next-word)
   (the-next-word)))
```

Так же как для предложения, в этот класс введены две дополнительные операции:

- * word-value Текущее значение слова
- * solved-p Имеет значение true (истина), если определены окончательно все буквы в данном слове

Теперь можно определить класс cipher-letter (буква шифра). Объекты этого класса относятся к «классной доске», являются предполагаемым и обладают свойством зависимости. Помимо этих свойств, такие объекты имеют значение (например, «Н») и список возможных способов соотнесения с буквами исходного текста. Опишем это следующим образом:

```
(defclass cipher-letter (blackboard-object assumable-object dependent)
  ((the-word :accessor the-word
              :initarg :the-word)
   (the-cipher-letter :accessor the-cipher-letter
                       :initarg :the-cipher-letter))
  ((the-plain-assumptions :accessor the-plain-assumptions)))
```

Слот с именем the-plain-assumption может содержать множество объектов, предполагаемых в качестве букв исходного текста. Это позволяет источникам знаний на основании предыстории сделанных и отвергнутых предположений избежать дальнейших ошибок.

В этот класс введена только одна дополнительная операция:

- | • letter-value | Текущее значение буквы |
|----------------|------------------------|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| A | 10 |
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |
| G | 16 |
| H | 17 |
| I | 18 |
| J | 19 |
| K | 20 |
| L | 21 |
| M | 22 |
| N | 23 |
| O | 24 |
| P | 25 |
| Q | 26 |
| R | 27 |
| S | 28 |
| T | 29 |
| U | 30 |
| V | 31 |
| W | 32 |
| X | 33 |
| Y | 34 |
| Z | 35 |

Необходимо помнить также о восьми операциях суперкласса *assumable-object*. Теперь обратимся к классу *alphabet* (алфавит). Этот класс содержит данные об алфавите исходного текста и шифра, а также о взаимосоответствии между ними. Это необходимо, чтобы источники знаний могли получить информацию о выявленных соответствиях между буквами шифра и текста и тех, которые еще предстоит найти. Например, если уже доказано, что буква

«С» в шифре соответствует букве «М» текста, то этот факт фиксируется в алфавите и источники знаний уже не будут делать других предположений в отношении буквы «М». Для эффективности обработки полезно иметь возможность получать данные о взаимосоответствии букв шифра и текста двумя способами: по букве шифра и по букве исходного текста. Определим класс alphabet следующим образом:

```
(defclass alphabet (blackboard-object assumable-object)
  ((the-plaintext-map :accessor the-plaintext-map)
   (the-ciphertext-map :accessor the-ciphertext-map)))
```

В отношении класса cipher-letter необходимо переопределить операции суперкласса assumable-object, чтобы использовать их применительно к алфавиту. Определим теперь класс «классная доска», который состоит из пяти описанных ранее классов:

```
(defclass blackboard ()
  ((the-assumptions :accessor the-assumptions)
   (the-sentence :accessor the-sentence)
   (the-words :accessor the-words)
   (the-letter :accessor the-letter)
   (alphabet :accessor the-alphabet
    :initform (make-instance 'alphabet))))
```

Для большинства слотов не определяются квалификаторы :initarg и :initform, поскольку в начальный момент «классная доска» не содержит никаких предположений, предложений, слов и букв. Однако в исходном состоянии «классная доска» содержит алфавит (без данных о соответствии шифр-тексту), чем определяется наличие квалификатора :initform в последнем слоте. Класс blackboard содержит операции state-assumption и retract-assumption, которые реализуются для всех экземпляров класса assumption, а также операции add-object и remove-object от класса blackboard-object. Определим еще пять операций, специфичных для «классной доски»:

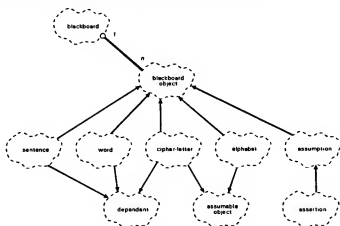


Рис. 11-2. Диаграмма классов «классной доски».

* reset	Очистка «классной доски»
* assert-problem	Помещение на доске содержания задачи
* solved-p	Имеет значение true (истина), если предложение расшифровано
* retrieve-solution	Значение исходного текста
* connect	Разрешение доступа к доске источника знаний

Последняя операция устанавливает дисциплину связи «классной доски» с источниками знаний. На рис. 11-2 приведена итоговая диаграмма классов, связанных с классом blackboard. Эта диаграмма в первую очередь отражает отношения наследования. Отношения использования для простоты опущены (например, между assumption и assumable-object).

Проектирование источников знаний

В предыдущем разделе мы выделили тринадцать источников знаний, относящихся к решаемой задаче. Теперь можно приступить к проектированию структур классов для этих видов знаний (как это было сделано для «классной доски») и обобщить их в более абстрактные классы.

Проектирование специализированных источников знаний. Предположим, что существует абстрактный класс knowledge-source (по аналогии с классом blackboard-object). Но, прежде чем определять все тринадцать источников в качестве подклассов одного общего суперкласса, нужно посмотреть не образуют ли знания группы из нескольких источников (кластеры). Действительно, такие группы имеют место: некоторые источники знаний оперирует целым предложением, другими — словами, фрагментами слов или отдельными буквами. Отразим этот факт в следующих определениях:

```
(defclass sentence-knowledge-source (knowledge-source)
  ())
(defclass word-knowledge-source (knowledge-source)
  ())
(defclass string-knowledge-source (knowledge-source)
  ())
(defclass letter-knowledge-source (knowledge-source)
  ())
```

Для каждого из приведенных абстрактных классов можно создать специфические подклассы. Подклассы от класса sentence-knowledge-source будут следующими:

```
(defclass sentence-structure-knowledge-source (sentence-knowledge-source)
  ())
(defclass solved-knowledge-source (sentence-knowledge-source)
  ())
```

А для класса word-knowledge-source следующими:

```
(defclass word-structure-knowledge-source (word-knowledge-source)
  ())
(defclass small-word-knowledge-source (word-knowledge-source)
  ())
(defclass pattern-matching-knowledge-source (word-knowledge-source pattern-matcher)
  ())
```

Последний класс требует некоторых пояснений. Ранее упоминалось, что цель такого класса состоит в проверке вариантов слов по шаблону. Для описания шаблона можно воспользоваться обозначениями, принятыми в операционной системе Unix:

* Любой элемент	?
* Отсутствие элемента	~
* Несколько элементов	*
* Начало группы	{
* Конец группы	}

С помощью этих символов покажем пример шаблона для данного класса ?E~{AEIOU}, означающий слово из трех букв, начинающееся с любой буквы, далее следует буква E и, наконец, любая буква, кроме гласных A, E, I, O, U. Поскольку проверка по шаблону является методом полезным как для данной системы в целом, так и в других областях приложения, соответствующий класс целесообразно выполнять в качестве смешиваемого, а не относить к одному из источников знаний. По тем же причинам класс `pattern-matcher` целесообразно выполнить в виде подкласса от более общего класса `dictionary` (словарь). Словарь является объектом для помещения в него слов, которые можно извлекать из словаря непосредственно, но объект `pattern-matcher` позволяет находить нужные слова путем фильтрации по заданному шаблону. Зафиксируем принятые решения в следующих определениях:

```
(defclass dictionary ()
  ())
(defclass pattern-matcher (dictionary)
  ())
```

Слоты этих двух классов не приведены, поскольку в данном случае это не существенно. Словарь следует рассматривать как список слов или поток с произвольным доступом — в любом случае поведение экземпляров данного класса одинаково. Продолжим определение подклассов для класса `string-knowledge-source`:

```
(defclass common-prefix-knowledge-source (string-knowledge-source)
  ())
(defclass common-suffix-knowledge-source (string-knowledge-source)
  ())
(defclass double-letter-knowledge-source (string-knowledge-source)
  ())
(defclass legal-string-knowledge-source (string-knowledge-source)
  ())
```

И наконец, определим подклассы для класса `letter-knowledge-source`:

```
(defclass direct-substitution-knowledge-source (letter-knowledge-source)
  ())
(defclass vowel-knowledge-source (letter-knowledge-source)
  ())
(defclass consonant-knowledge-source (letter-knowledge-source)
  ())
(defclass letter-frequency-knowledge-source (letter-knowledge-source)
  ())
```


Обобщение источников знаний. Для всех упомянутых специализированных классов определены только две операции:

- * reset Повторный запуск источника знаний
- * evaluate-blackboard Определение состояния «классной доски»

Причина столь упрощенного интерфейса в относительной автономности знаний: мы указываем на интересующий объект «классной доски» и даем команду выполнить соответствующие правила по оценке состояния всей доски в целом. При выполнении правил оценки каждый из источников знаний может осуществить следующие действия:

- * Сделать предположение относительно истинного значения одной из букв шифра.
- * Найти противоречие в отношении ранее сделанных предположений и отклонить такие предположения.
- * Доказать правильность какого-либо предположения.
- * Сообщить контроллеру о наличии полезной информации для вынесения на доску.

Все эти действия являются общими для всех источников знаний. Вообще говоря, все приведенные операции образуют механизм вывода заключений. Определим механизм вывода как объект, который на основании известных правил либо производит действия согласно правилам для перехода к новым правилам (прямая цепь), либо принимает решение по выдвинутым ранее гипотезам (обратная цепь). На основании сказанного введен следующий класс:

```
(defclass inference-engine ()
  ((the-rules :accessor the-rules :initarg :the-rules)))
```

Слот, определенный в этом классе, включает в себя все правила механизма вывода. Лишь одна операция определена первично в этом классе — (evaluate-rubs) обработка правил механизма вывода. Таким образом, основная цель специализированных источников знаний состоит в возможности хранить соответствующие правила. Операция evaluate-blackboard в первую очередь обращается к методу evaluate-rules, а затем переходит к выполнению одной из четырех упомянутых выше операций.

Что такое правило? Для иллюстрации приведем следующее правило, касающееся знаний о суффиксах:

```
? ' (( * I ? ? )
      (* I N G)
      (* I E S)
      (* I E D))
```

Это правило означает, что заданному шаблону для строки букв *I?? (условие) могут соответствовать суффиксы ING, IES и IDE (заключение). На основании этого можно определить класс для представления правил:

```
(defclass rule ()
  ((the-antecedent :accessor the-antecedent)
   (the-consequent :accessor the-consequent)))
```

С точки зрения строения данного класса можно утверждать, что источники знаний являются разноречивостью механизма вывода. А с точки зрения

зависимости источников знаний от объектов на «классной доске» можно утверждать, что источники знаний обладают качеством зависимости. Поэтому механизм зависимости вводится в источники знаний по аналогии с объектами «классной доски».

Определим это следующим образом:

```
(defclass knowledge-source (inference-engine dependent)
  ((the-blackboard :accessor the-blackboard)
   (the-controller :accessor the-controller)
   (the-assumptions :accessor the-assumptions)))
```

В этом классе имеются слоты the-blackboard и the-controller для связи с объектами системы. Слот the-assumption необходим для хранения информации о последовательности сделанных предположений, чтобы избежать повторения и учесть ошибки.

Экземпляры класса knowledge-source содержат в себе объекты «классной доски». Необходимо ввести класс knowledge-source, охватывающий перечень всех источников знаний, относящихся к решаемой задаче. На основании сказанного запишем:



Рис. 11-3. Диаграмма классов для источников знаний.

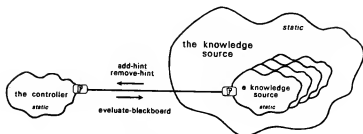


Рис. 11-4. Механизм контроллера.

```
(defclass knowledge-sources ()
  ((the-knowledge-sources
    :accessor the-knowledge-sources
    :initform (make-knowledge-sources))))
```

Наличие квалификатора `:initform` объясняется тем, что при создании объекта класса `knowledge-source` создается 13 объектов специализированных источников знаний. Для класса `knowledge-source` определяются три операции:

- * `restart` Перезапуск источника знаний
- * `start-knowledge-sources` Установка начальных значений условий для каждого из источников знаний
- * `connect` Установление связи источника знаний с «классной доской» или с контроллером

На рис. 11-3 показана структура классов для созданных в процессе проектирования классов `knowledge-source`.

Проектирование контроллера

На рис. 11-4 показано взаимодействие контроллера с отдельными источниками знаний. В процессе поэтапной расшифровки криптограммы отдельные источники знаний могут выявлять полезную информацию для сообщения контроллеру. И наоборот, может быть обнаружено, что ранее переданная информация оказалась ложной и ее надо устранить. Поскольку все источники знаний имеют равные права, контроллер должен выбрать из них тот, информация которого может оказаться наиболее полезной, и активизировать его вызовом операции `evaluate-blackboard`.

Как определяет контроллер, какой из источников знаний следует активизировать? Можно предложить несколько полезных правил:

- * Большой приоритет имеет окончательное утверждение перед допущением.
- * Наибольший вес имеет информация источника, отвечающего за всю фразу.
- * Проверка по шаблону более приоритетна, чем обработка структуры предложения.

Сообщение контроллеру можно реализовать в классе, определенном для зависимостей. Однако в виду наличия приоритетов сообщений лучше образовать соответствующий подкласс:

```
(defclass ordered-dependent (dependent)
  ())
```

Объекты нового класса почти такие же, как объекты класса `dependent`, но операция `add-dependency` выполняется согласно приоритетам. Теперь можно определить интерфейс для класса, означающего контроллер:

```
(defclass controller ()
  ((the-hints
    (the-knowledge-sources
      :accessor the-hints)
      :accessor the-knowledge-sources)))
```

Слот с именем the-hints означает объект класса ordered-dependent. Для объекта-контроллера определяется семь операций:

- * reset Перезапуск контроллера
- * add-hint Добавить данные от источника знаний
- * remove-hint Удалить данные от источника знаний
- * process-next-hint Найти данные с наибольшим приоритетом
- * solved-p Селектор: имеет значение true (истина), если задача решена
- * unable-to-procesed-p Селектор: имеет значение true (истина), если источник знаний зашел в тупик
- * connect Устанавливает связь контроллера с источником знаний

11.3. ОБЪЕДИНЕНИЕ В СИСТЕМУ

Объединение элементов «классной доски»

Теперь, когда ключевые абстракции предметной области выявлены, можно приступить к их соединению в действующую систему. Поскольку CLOS не является строго типированным языком, допустимо (и желательно) вести разработку элементов системы последовательно-итеративно. Мы будем реализовывать и проверять компоненты системы по отдельности.

Объединение объектов верхнего уровня. На рис. 11-5 показана диаграмма объектов нашей системы на самом верхнем уровне, которая полностью соответствует структуре метода «классной доски», приведенной на рис. 11-1. В качестве класса, отвечающего данному проектному решению, используем следующий класс, названный cryptographer:

```
(defclass cryptographer ()
  ((the-blackboard
    (the-knowledge-sources
      (the-controller
        :accessor the-blackboard
        :initform (make-instance 'blackboard))
        :accessor the-knowledge-sources
        :initform (make-instance 'knowledge-source))
        :accessor the-controller
        :initform (make-instance 'controller))))
```

Для всех слотов этого класса использован квалификатор :initform для инициализации всех составляющих класс объектов. Мы отказались от использования в слотах квалификатора :type, хотя тип всех слотов известен, потому, что квалификатор :initform гарантирует нас от возможных ошибок. Кроме того, в языке CLOS контроль соответствия типов выполняется только во время исполнения программы.

Для введенного класса определяются три исходных операции:

- * initialize-istance Объединение «классной доски», источников знаний и контроллера в общий объект
- * restart Повторный запуск задачи
- * decipher Процесс дешифровки заданной криптограммы

Для стиля программирования CLOS характерно определять функцию, которая генерирует объекты класса cryptographer. Определим ее так:

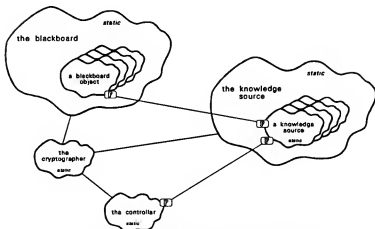


Рис. 11-5. Диаграмма объектов для задачи дешифровки.

```
(defun create-cryptographer ()
  (defvar *cryptographer* (make-instance 'cryptographer)))
```

Вызов этой функции создает переменную `*cryptographer*`, которая означает экземпляр класса `cryptographer`. Использование макроопределения `defvar` объясняется тем, что переменная `*cryptographer*` должна определяться в качестве глобальной.

В процессе инициализации объекта реализуются связи между «классной доской» и источниками знаний, а также между источниками знаний и контроллером. Операцию `initialize-instance` нет необходимости вызывать специально, поскольку она относится к разряду первичных методов и вызывается автоматически с помощью `make-instance`. Поэтому лучше всего определять связи между элементами системы с помощью квалификатора `:after` в методе `initialize-instance` следующим образом:

```
(defmethod initialize-instance :after ((application cryptographer))
  (connect (the-blackboard application) (the-knowledge-sources application))
  (connect (the-knowledge-sources application) (the-controller application)))
```

Обратим внимание на способ применения методов доступа к слотам основного объекта. Метод `reset` предельно прост: его цель — вернуть в исходное состояние все слоты объекта:

```
(defmethod reset ((application cryptographer))
  (reset (the-blackboard application))
  (reset (the-knowledge-sources application))
  (reset (the-controller application))
  t)
```

Уточним, что метод `decipher` принимает строку в качестве исходной криптограммы. Это позволяет выполнить механизм дешифровки независимо от способа ввода задания; для нас желательно создавать обобщенные механизмы решения, не отвлекаясь на специфику конкретного задания. Теперь мы можем определить основную функцию нашей задачи по дешифровке:

```
(defun decode ()
  (reset *cryptographer*)
  (format t "~&Enter the ciphertext =>")
  (let ((plaintext (decipher *cryptographer* (string-upcase (read-line)))))
    (cond
      ((equal "" plaintext)
       (format t "~&Unable to decode the ciphertext")
       ()
      (t
       (format t "~&Plaintext => ~a" plaintext)
       ())))))
```

Для объявления локальной переменной plaintext использована форма оператора let, позволяющего возвращать значение, получаемое методом decipher. Для упрощения операций с источниками знаний выполняется преобразование исходной строки шифра к верхнему регистру алфавита.

Метод decipher несколько сложнее. В первую очередь с помощью метода assert-problem задание помещается на «классную доску». После этого активизируются источники знаний. И наконец, начинается циклический процесс обращения источников знаний к контроллеру с новыми и новыми предположениями и выводами до тех пор, пока не будет найдено решение задачи, либо до исчерпания источников знаний. В тексте программы это будет выглядеть так:

```
(defmethod decipher ((application cryptographer) ciphertext)
  (check-type ciphertext string)
  (let ((controller (the-controller application)))
    (assert-problem (the-blackboard application) ciphertext)
    (start-knowledge-sources (the-knowledge-sources application))
    (loop
      (when (unable-to-proceed-p controller) (return ""))
      (when (solved-p controller (retrieve-solution controller)))
        (process-next-hint controller))))))
```

В приведенном методе имеются два аргумента с именами application и ciphertext. Первый аргумент для отличия квалифицирован классом cryptographer. Поскольку этот метод не является множественным (имея в виду, что он отличается не только аргументом), мы не квалифицируем второй аргумент, хотя для надежности следует проверить его тип.

Теперь посмотрим внимательно на один из методов «классной доски» и на один из методов контроллера, а именно assert-problem и retrieve-solution.

Метод assert-problem интересен тем, что он создает структуру объектов «классной доски». По ранее указанным причинам обрабатывать строку удобнее справа налево. Используя псевдокод, опишем наш алгоритм следующим образом:

```
:: убираем из строки все пробелы в начале и в конце
:: если получилась пустая строка, то возврат
:: создаем объект-предложение
:: выносим предложение на доску
:: создаем объект-слово (самое крайнее справа)
:: добавляем слово к предложению
:: выносим слово на доску
:: циклически для каждого символа справа-налево
::   если пробел
::     делаем следующее слово текущим
::     создаем объект-слово
```

```

;;      добавляем слово к предложению
;;      выносим слово на доску
;;
;;      иначе
;;      создаем объект-букву шифра
;;      добавляем букву к слову
;;      выносим букву на доску

```

На языке CLOS этот алгоритм будет выражен следующим образом:

```

(defmethod assert-problem ((the-blackboard blackboard) (ciphertext))
  (check-type ciphertext string)
  (let ((the-string (string-trim '("#\space") ciphertext))
        (current-word ())
        (next-word ())
        (current-letter ()))
    (when (string-equal "" the-string) (return-from assert-problem nil))
    (setf (the-sentence the-blackboard) (make-instance 'sentence))
    (setf current-word
      (make-instance 'word
        :the-sentence (the-sentence the-blackboard)
        :the-next-word next-word))
    (push current-word (the-words (the-sentence the-blackboard)))
    (push current-word (the-words the-blackboard))
    (dotimes (index (length the-string) 1)
      (let ((the-character
            (char the-string (- (- (length the-string) 1) index))))
        (if (eq '#\space the-character)
            (and (setf next-word current-word)
                 (setf current-word)
                 (make-instance 'word
                   :the-sentence (the-sentence the-blackboard)
                   :the-next-word next-word))
            (setf (the-previous-word next-word) current-word)
            (push current-word (the-words (the-sentence the-blackboard)))
            (push current-word (the-words the-blackboard)))
        (and (setf current-letter
              (make-instance 'cipher-letter
                :the-word current-word
                :the-cipher-letter the-character))
             (push current-letter (the-letters current-word))
             (push current-letter (the-letters the-blackboard))))))
    1))

```

Метод `assert-problem`, подобно методу `decipher`, не является множественным, поэтому в нем квалифицируется только первый аргумент. Гораздо проще выглядит метод `retrieve-solution`, который только возвращает текст предложения:

```

(defmethod retrieve-solution ((the-blackboard blackboard))
  (sentence-value (the-sentence the-blackboard)))

```

Чуть сложнее метод `sentence-value`, который формирует предложение из слов разделенных пробелами:

```

(defmethod sentence-value ((the-sentence))
  (let ((the-string ()))
    (dotimes (a-word (the-words the-sentence) 1)
      (setf the-string

```

```
(concatenate 'string the-string (word-value a-word) <=>))
(string-right-trim '#space) the-string)))
```

Аналогичным образом метод `word-value` создает слово из отдельных букв:

```
(defmethod word-value ((the-word word))
  (let ((the-string '))
    (dolist (a-letter (the-letters the-word) t)
      (push (letter-value a-letter) the-string)
      (reverse coerce the-string 'string))))
```

Еще несколько сложнее метод `letter-value`. Он возвращает значение самого последнего предположения, сделанного относительно данной буквы (если таковые имеются), либо исходное значение этой буквы:

```
(defmethod letter-value ((the-letter cipher-letter))
  (if (null (the-plain-assumptions the-letter))
      (the-cipher-letter the-letter)
      (the-plain-letter (first (the-plain-assumptions the-letter)))))
```

Реализация механизма выдвижения предположений. Нам нужно создать механизм, который формирует и отменяет значения элементов, имеющих на «классной доске». Но сначала рассмотрим механизм выдвижения предположений об этих значениях. Этот механизм интересен ввиду его динамичности. В процессе поиска решения предположения непрерывно создаются и разрушаются.

На рис. 11-6 показаны главные потоки управления в процессе выдвижения предположений. Источник знаний сообщает о имеющихся предположениях «классной доске», которая определяет к каким элементам это предположение относится.

Полезно посмотреть на этот механизм прежде всего в отношении наиболее специфичных объектов. Предположение относительно буквы шифра реализуется предельно просто, путем занесения его в соответствующий список предположений. Это выглядит следующим образом:

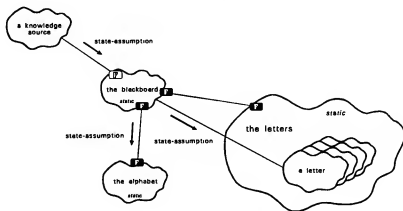


Рис. 11-6. Механизм выдвижения предположений.


```
(defmethod state-assumption (the-assumption (the-object cipher-letter))
  (check-type the-assumption assumption)
  (push the-assumption (the-assumption the-object))
  t)
```

Этот метод отличается только по второму аргументу, поэтому первый аргумент не квалифицирован. Для класса `assumption` логика задачи требует хранения данных о всех объектах, связанных с выдвинутыми предположениями. Для этой цели служит слот `the-assumable-objects`. Например, если сделано предположение о какой-либо букве шифра, то эта буква включается в список объектов-предположений. Поскольку предположения могут быть связаны с алфавитом, можно ввести в класс `assumable-object` обобщенный метод с квалификатором `:after` в следующем виде:

```
(defmethod state-assumption :after (the-assumption (the-object assumable-object))
  (push the-object (the-assumable-object the-assumption)))
```

Это хороший пример использования метода с квалификатором: если некоторая группа операций в обобщенной функции может оказаться связанной либо с более общим, либо с более специализированным классом, то эти операции целесообразно снабдить одним из квалификаторов `:after`, `:before` или `:around`.

Что если допущение в отношении какой-либо буквы уже доказано? В этом случае следует прекратить выдвижение новых предположений. Чтобы остановить любые действия, которые могут изменить существующее состояние, определим следующий метод с квалификатором `:around`:

```
(defmethod state-assumption :around (the-assumption (the-object cipher-letter))
  (if plain-letter-asserted-p the-object (the-plain-letter the-assumption))
  nil
  (call-next-method)))
```

Квалификатор `:around` в отличие от `:before` позволяет принять решение о том, следует ли вообще вызывать основной метод (с помощью `call-next-method`). Приведенный выше метод в случае окончательного доказательства значения буквы возвращает значение `nil`. Можно переопределить метод `plain-letter-asserted-p` так, чтобы он не предшествовал всем объектам-предположениям. Для класса `cipher-letter`, в частности, определим его реализацию так:

```
(defmethod plain-letter-asserted-p ((the-object cipher-letter) the-letter)
  (check-type the-letter character)
  (and (not (null (the-plain-assumptions the-object)))
       (not (retractable-p (first (the-plain-assumptions the-object))))))
```

Этот метод возвращает значение `true` (истина), только если существуют какие-либо предположения и последнее из них не является доказанным. Теперь нужно уточнить метод `retractable-p` для классов `assumption` и `assertion`:

```
(defmethod retractable-p ((the-assumption assumption))
  t)
(defmethod retractable-p ((the-assumption assumption))
  nil)
```

Выдвижение предположений относительно алфавита требует совершенно иначе определить основной метод и метод с квалификатором `:around`, по

сколько объекты `alphabet` организованы принципиально иначе, чем буквы шифра. Метод с квалификатором `:around`, в частности, позволяет проверить является ли предположение о букве шифра или букве текста уже доказанным:

```
(defmethod state-assumption :around (the-assumption (the-object alphabet))
  (if (or (plain-letter-asserted-p the-object
    (the-plain-letter the-assumption))
    (cipher-letter-asserted-p the-object
    (the-cipher-letter the-assumption)))
    nil
    (call-next-method)))
```

Основной метод является более сложным и может быть выражен на псевдокоде так:

```
:: установить эквивалент для буквы исходного текста
:: установить эквивалент для буквы шифра
:: образовать новую связь текст/шифр и внести ее в состав предположений о тексте
:: образовать новую связь шифр/текст и внести ее в состав предположений о шифре
```

На языке CLOS это можно выразить таким образом:

```
(defmethod state-assumption (the-assumption (the-object alphabet))
  (check-type the-assumption assumption)
  (let ((plain (the-plain-letter the-assumption))
        (cipher (the-cipher-letter the-assumption))
        (plain-map (the-plaintext-map the-object))
        (cipher-map (the-ciphertext-map the-object)))
    (if (plain-letter-defined-p the-object plain)
        (push the-assumption (second (assoc plain plain-map)))
        (setf (the-plaintext-map the-object)
              (acons plain (list (list the-assumption)) plain-map)))
    (if (cipher-letter-defined-p the-object cipher)
        (push the-assumption (second (assoc cipher cipher-map)))
        (setf (the-ciphertext-map the-object)
              (acons cipher (list (list the-assumption)) cipher-map)))
    t))
```

В этом методе необходима проверка того, является ли та или другая буква уже определенной. В этом случае просто образуется новая связь букв. При отсутствии такой неопределенности необходимо инициализировать исходное значение связей текст/шифр и шифр/текст. Эти подробности не являются существенными для объектов-пользователей, поэтому алфавит (т.е. два списка связей букв) ограничен для доступа (доступ ведется через методы абстракций более высокого уровня). Выполним защиту указанию описания в следующей форме:

```
(defmethod plain-letter-asserted-p ((the-object alphabet) the-letter)
  (check-type the-letter character)
  (let ((the-association (assoc the-letter (the-plaintext-map the-object)))
        (and (not (null the-association))
              (not (retractable-p (first (second the-association)))))))
    the-association)
(defmethod cipher-letter-asserted-p ((the-object alphabet) the-letter)
  (check-type the-letter character)
  (let ((the-association (assoc the-letter (the-ciphertext-map the-object)))
        (and (not (null the-association))
              (not (retractable-p (first (second the-association)))))))
    the-association)
```

Пробуем еще более обобщить эту операцию. Для источников знаний существенно знать, является ли некоторое множество букв частично или полностью определенным. Предположим такую форму определения:

```
(some-letter-p 'plain-letter-asseried-p
               (the-letters the-blackboard)
               '($\A $\E $\I $\O $\U))
```

Эта запись возвращает значение true (истина), если определена хотя бы одна из гласных букв. Можно оформить это же описание в виде макроопределения:

```
(defmacro some-letter-p (text source letters)
  (list 'dolist (list 'item letters 'nil)
        (list 'if (list text source 'item)
                (list 'return 't))))
```

Модифицировав это макроопределение, можно определить операцию проверки соответствия для целой группы букв:

```
(defmacro all-letter-p (text source letters)
  (list 'dolist (list 'item letters 't)
        (list 'if (list 'not (list text source 'item)
                        (list 'return 'nil)))))
```

Теперь мы готовы дать определение метода state-assumption для «классной доски» в целом. Сначала изложим его на псевдокоде:

```
::      внести предположение в список предположений
::      определить это предположение для алфавита доски
::      для всех букв «классной доски»
::      установить данное предположение
```

На языке CLOS этому соответствует запись:

```
(defmethod state-assumption (the-assumption (the-blackboard blackboard))
  (check-type the-assumption assumption)
  (push the-assumption (the-assumption the-blackboard))
  (state-assumption the-assumption (the-alphabet theblackboard))
  (dolist (item (the-letter the-blackboard) t)
    (if (equal (the-cipher-letter item) (the-cipher-letter the-assumption))
        (state-assumption the-assumption item))))
```

В простейшем случае отмена предположения выражается в прерывании метода state-assumption. Например, чтобы отменить предположение о букве шифра, мы просто очищаем список предположений для исходного текста по данному (отвергнутое) предположение включительно.

При этом ликвидируются и все предположения, построенные на ошибочном допущении. Можно создать и более эффективный механизм. Допустим, что сделано предположение, согласно которому однобуквенное слово соответствует букве I (по некоторым причинам требуется гласная). Далее сделано предположение, что другому двухбуквенному слову соответствует NN (нужны согласные). Если первое предположение окажется ошибочным, то второе вполне может быть сохранено. При таком подходе класс assumption нужно дополнить еще одним слотом, в котором регистрируется связь предположений между собой (взаимная зависимость).

Подключение новых источников знаний

Теперь, когда определены ключевые абстракции «классной доски» и механизмы выдвижения и проверки предположений, необходимо реализовать механизм вывода (класс inference-engine), связывающий все источники знаний в единое целое. Ранее уже упоминалось, что механизм вывода должен реализовать одну основную операцию, а именно обработку правил. Мы не будем на этом подробно останавливаться, поскольку этот метод не несет в себе принципиально новых решений. Убедившись в правильной работе механизма вывода, можно последовательно вводить в систему отдельные источники знаний. Целесообразность последовательного процесса введения знаний объясняется двумя причинами:

- Трудно заранее выяснить, какие правила существенны для каждого из источников знаний, не испытав систему на конкретной задаче.
- Отладка базы знаний существенно упрощается при включении в нее правил порциями.

Реализация источников знаний является предметом инженерии знаний и требует консультаций с эксперта (это могут быть сами разработчики системы или криптологи). При анализе источников знаний может выявиться, что одни правила бесполезны, другие слишком специализированы или излишне обобщены, а некоторых правил недостает. После такого анализа правила источника знаний могут модифицироваться, а может потребоваться создание нового источника знаний.

На рис. 11-7 показан механизм взаимодействия отдельных источников знаний с объектами «классной доски». Отметим, что каждый источник знаний управляет только списком относящихся к нему объектов (через slot the-references, наследованный от суперкласса dependent).

Кроме того, каждый источник знаний связан с «классной доской» и может получать данные о состоянии всех объектов на доске. При активизации источника знаний с помощью контроллера (см. рис. 11-4) начинается действие соответствующего механизма вывода по обработке связанных с ним объектов. Это действие выражается в выдвижении и отмене предположений на основе механизма, который показан на рис. 11-6.

В процессе реализации источников знаний могут выявиться общие для нескольких источников правила и (или) поведение. Например, источник знаний о структуре слов и источник знаний о структуре предложения могут иметь в своем составе общие правила относительно порядка отдельных букв и слов. В обоих случаях существо правил одно и то же, поэтому целесообразно ввести новый класс — источник знаний о структуре — и использовать его путем смешения для получения общего эффекта.

Такое изменение структуры классов подчеркивает тот факт, что процесс обработки правил определяется не только источниками знаний, но и характером объектов «классной доски». Например, один из источников знаний может реализовывать прямую цепь выводов в отношении одних объектов и обратную цепь в отношении других. Более того, различные источники знаний могут по-разному оперировать с одним и тем же объектом. Это дает повод перейти к более сжатому написанию методов на основе множественного полиморфизма, используемого в языке CLOS применительно к множественным методам. Возьмем для примера следующую обобщенную функцию:

```
(defgeneric evaluate-rules
  (the-knowledge-source the-blackboard-object))
```

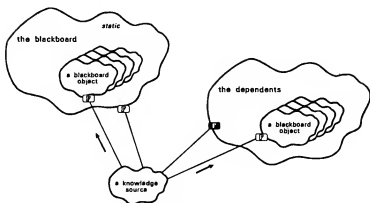


Рис. 11-7. Механизм действия источников знаний.

Можно сделать множественный метод специализированным по отношению к виду источника знаний и к классу объектов доски:

```
(defmethod evaluate-rules
  ((the-structure-knowledge-source structure-knowledge-source)
   (the-blackboard-object blackboard-object))
  ...)
```

Этот метод специализирован в отношении класса `structure-knowledge-source`, но от вида объектов «классной доски» не зависит. Напротив, следующий метод:

```
(defmethod evaluate-rules
  ((the-small-word-source small-word-knowledge-source)
   (the-word word))
  ...)
```

является зависимым как от класса `small-word-knowledge-source`, так и от `word`.

11.4. МОДИФИКАЦИЯ

Расширение функциональных возможностей

В данном разделе мы попытаемся улучшить возможности проектируемой системы и определить ее чувствительность к вносимым изменениям. В интеллектуальных системах очень важно наряду с решением задачи получать информацию о ходе этого решения. Для этого нужно придать системе качество самоанализа: регистрировать ход активизации источников знаний, причины и характер выдвигаемых предположений и т.д. (чтобы иметь возможность запросить у системы, по какой причине сделано конкретное предположение и к каким результатам оно приводит).

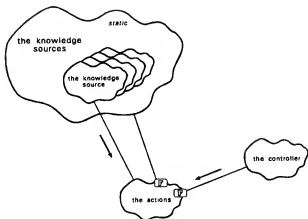


Рис. 11-8. Механизм объяснения.

Для реализации такого свойства необходимо сделать две вещи. Во-первых, нужно ввести механизм трассировки действий контроллера и источников знаний, а во-вторых, модифицировать некоторые методы, чтобы они отмечали соответствующую информацию. Общая схема механизма самоанализа (механизма объяснения) приведена на рис. 11-8. Суть механизма состоит в наличии некоторого общего регистратора всех действий источников знаний и контроллера. Посмотрим какие классы могут понадобиться для реализации механизма объяснения.

Во-первых, для регистрации всех указанных выше действий введем класс action:

```
(defclass action ()
  ((who :accessor who      :inform :who)
   (what :accessor what    :inform :what)
   (why :accessor why      :inform :why)))
```

Экземпляр данного класса создается, например, при активизации контроллером какого-либо источника знаний. При этом в слот who (кто) заносится указание на контроллер, в слот what (что) — на источник знаний, а в слот why (почему) — какое-либо пояснение (например, приоритет сообщения или предположения). Для сбора и хранения объектов класса action нужен другой класс actions:

```
(defclass actions ()
  ((the-actions :accessor the-actions)))
```

Этот класс должен входить в структуру класса cryptographer. Первая часть нашего нового механизма сделана, и вторая не будет сложной. Посмотрим на то, где в нашей системе происходят основные события. Это может происходить в следующих пяти местах:

- * методы выдвижения предположений
- * методы отмены предположений
- * методы активизации источников знаний
- * методы обработки правил
- * методы регистрации решений, получаемых от источников знаний

Удобно использовать при этом методы с квалификаторами `:before` и `:after`. Возьмем для примера обобщенную функцию `state-assumption`, которая специализирована для букв шифра и алфавита. Для регистрации событий в этом методе воспользуемся квалификатором: `after`:

```
(defmethod state-assumption :after (the-assumption (the-object assumable-object))
  (let ((the-action (make-instance 'action
                                   :who the-object
                                   :what the-assumption
                                   :why ())))
    (push the-action (the-actions *cryptographer*)))))
```

Нам не придется изменять код исходного основного метода, мы только уточняем поведение с помощью квалификаторов. Для полиоты нам остается только создать объект, отвечающий на вопросы пользователя системы (кто? что? когда? почему?). Спроектировать такой объект не сложно, поскольку вся нужная для его работы информация может быть получена от объектов класса `actions`.

Изменение технических требований

Новые технические требования к системе могут быть удовлетворены при минимальных изменениях принятых проектных решений. Допустим, что предъявлено три вида новых требований к данной системе:

- * возможность дешифровки других языков (кроме английского)
- * возможность дешифровки шифра, использующего неоднозначные подстановки букв
- * способность самообучения

Первое требование самое простое, поскольку связь нашей системы с английским языком не является существенной. Эта связь отражается только на правилах источников знаний. Даже класс «алфавит» сделан независимым от конкретного набора букв. Второе требование существенно сложнее, но разрешимо в рамках механизма «классной доски». Это потребует введения новых источников знаний относительно подстановки букв. Ключевые механизмы и абстракции при этом также сохраняются, но потребуются введение новых классов, которые будут действовать в рамках существующих механизмов. Самым трудным является последнее требование, так как обучение машины относится к области искусственного интеллекта. Можно, например, предложить контроллеру в тупиковых ситуациях обращаться за помощью к пользователю системы. Такие подсказки могут регистрироваться системой и позволяют в дальнейшей работе избегать подобных тупиков. Такой простейший механизм обучения может быть введен в нашу систему без существенного изменения структуры классов и в пределах действующих механизмов.

Дополнительная литература

Englemore, Morgan [C, 1988] изложили исчерпывающие требования и подходы к системам на основе «классной доски», включая их создание, теорию, проектирование и примеры применения. Есть два конкретных примера такой системы BBI (проект Stanford) и BLOB (министерство обороны Великобритании). Полезная информация по «классной доске» имеется у Hayes-Roth [J, 1985] и Nil [J, 1986]. Прямые и обратные цепочки вывода подробно рассмотрены для систем основанных на базах правил у Bart и Feigenbaum [J, 1981]; Brachman и Levesque [J 1985]; Hayes-Roth, Waterman, Lenat [J, 1983]; Winston, Horn [G, 1989]. Meyer и Matyas [I, 1982] подробно проанализировали разновидности шифров с точки зрения их дешифровки. Краткое описание CLOS с примерами можно найти в приложении.

Глава 12

Ada. Система управления движением

Индустрия создания программного обеспечения достигла к настоящему моменту такого уровня развития, что появилась возможность внедрить средства автоматизации в множество новых областей, в диапазоне от микрокомпьютерного управления двигателем автомобиля до освобождения людей от значительной части рутинной работы при съемках кинофильмов и комплектовании экипажей космических кораблей. Отличительной особенностью больших систем является их чрезвычайно высокая сложность. Конечно, достойным уважением попытки упростить реализацию таких систем. Однако практика показывает, что сложности, возникающие при реализации больших систем, огромны. В крупных проектах нередко бывают задействованы сотни программистов, которым необходимо написать миллион и более строк программ. При этом они должны учитывать множество требований, которые неизбежно меняются в процессе работы. Как правило, в рамках таких проектов создается не одна программа, работающая на одном компьютере, а комплекс программ, работающих в параллельной распределенной среде на большом числе связанных между собой компьютеров. Для того чтобы уменьшить вероятность неудачи при реализации большого проекта, обычно выбирают централизованный принцип управления. Данная организация работ обеспечивает высокую надежность как при разработке архитектуры системы, так и при объединении ее частей. Некоторые части системы выполняются по субконтрактам с другими компаниями.

Группа разработчиков никогда не собирается вместе, так как, во-первых, люди работают в разных местах, и, во-вторых, происходит постоянная текучка кадров.

Если за создание большой системы возьмется разработчик, который занимался написанием небольших программ, рассчитанных на одного пользователя, он столкнется с проблемой неопределенности и изменчивости принимаемых решений. Возможно, он решит, что глупо пытаться делать такую систему. Но деловой и научный мир остро нуждается в больших системах. Вообразим себе использование ручной системы для управления авиационными полетами вокруг столичного центра, управление жизнеобеспечением членов космического корабля или отчетную деятельность международного банка. Успешная автоматизация таких систем приводит не только к решению видимых проблем, но также приносит множество осязаемых и неосозаемых выгод, таких, как низкая операционная стоимость, высокая надежность, увеличение функциональных возможностей. Конечно же, ключевое слово здесь — *успешная*. Ясно, что создание больших систем — чрезвычайно трудная задача. Поэтому при ее решении необходимо применять все лучшее из инженерной практики и использовать интуицию ведущих проектировщиков. В этой главе представлена задача для демонстрации того, как объектно-ориентированное проектирование облегчает программирование больших проектов. В качестве языка программирования мы используем язык Ada, который является объектным, а не объектно-ориентированным, так как в нем не поддерживается механизм наследования.

Требования к системе управления движением

Система управления движением выполняет две главные функции: выбор маршрутов перевозок и контроль за транспортной системой. Эти функции включают планирование перевозок, контроль за перевозками, предотвращение конфликтов, прогнозирование неудач и регистрацию обслуживания. На рис. 12-1 приведена диаграмма основных элементов системы управления движением.

Система сбора и отображения информации на локомотиве состоит из множества дискретных и аналоговых датчиков для контроля таких параметров, как температура и давление масла, количество топлива, напряжение и сила тока на генераторе, число оборотов в минуту вала двигателя, температура воды, мощность тягового стержня. Значения с датчиков поступают к инженеру поезда через дисплейную систему, а к диспетчеру и обслуживающему персоналу, который находится вне поезда, через сеть передачи данных. Предупреждение или сигнал тревоги выбираются и регистрируются всякий раз, когда показания любого датчика выходят за пределы нормального режима. Регистрация показаний датчиков используется при проведении эксплуатационных работ и управлении расходом топлива.

Система управления двигателем сообщает в реальном времени инженеру поезда, как наиболее эффективно дросселировать и тормозить двигательные установки. Входными данными для этой системы являются: профиль и качество пути, ограничение по скорости, допустимые режимы, загрузка поезда и максимальная мощность. Исходя из этих данных, система может определить оптимальный по расходу топлива режим дросселирования и торможения двигательных установок, согласующийся с заданными режимами и требованиями безопасности. Рекомендации по дросселированию и торможению двигательных установок, профиль и качество пути, местоположение и скорость поезда могут отображаться на бортовой дисплейной системе.

Бортовая дисплейная система обеспечивает человеко-машинный интерфейс. На нее может выводиться информация с системы сбора и отображения информации локомотива, системы управления двигателем и устройства управления данными. Программируемые клавиши позволяют инженеру переключаться на различные дисплеи.

Устройство управления данными предоставляет сервис шлюза между всеми бортовыми системами поезда и сетью передачи данных, к которой подключены все поезда, диспетчеры и прочие пользователи. Отслеживание маршрутов движения поездов осуществляется с помощью двух устройств, подключенных к сети передачи данных: транспондеров местоположения и глобальной спутниковой системы определения местоположения Навстар (GPS). Система сбора и отображения информации на локомотиве может определять положение локомотива с помощью счетчика, подсчитывающего число оборотов колеса. Эта информация извлекается из транспондеров местоположения, которые размещены через каждый километр пути и более часто на опасных участках. Транспондеры передают свою идентификацию на устройство управления данными, из которой можно более точно определить местоположение поезда. Кроме того, поезд может быть подключен к приемникам GPS, с помощью которых положение поезда может быть определено с точностью до одного метра.

Околопутевые интерфейсные устройства размещаются там, где есть какое-либо управляемое устройство (например, переключатель), или датчик (например, инфракрасный датчик для обнаружения перегрева колес поезда). Каждое околопутевое интерфейсное устройство получает команды от локального контроллера наземного терминала (например, такне, как включить и выключить сигнал). Устройства могут быть отключены с помощью местной системы управления. Кроме того, каждое устройство может сообщать свои установочные параметры. Наземный терминал транслирует информацию с околопутевых интерфейсных устройств на проходящий мимо поезд и обратно. Контроллеры наземных терминалов расположены вдоль железнодорожного пути через такие расстояния, что любой поезд всегда находится в зоне действия не менее одного из них.

Каждый контроллер наземного терминала передает свою информацию на объединенную систему управления сетью. Связь между системой управления сетью и контроллером наземного терминала может осуществляться микроволновой связью, по наземным линиям или по оптоволоконной связи в зависимости от удаленности данного контроллера. Система управления сетью обеспечивает жизнеспособность всей сети. Она может автоматически направить информацию по другому пути в сети, если на прежнем пути произойдет отказ оборудования.

Система управления в свою очередь подсоединяется к одному или нескольким диспетчерским центрам, которые составляют систему управления железной дорогой, и к другим пользователям. В системе управления железной дорогой диспетчеры могут устанавливать маршруты поездов и колес для движения отдельных поездов. Отдельный диспетчер управляет различными территориями; каждая диспетчерская управляющая консоль может быть подключена для управления одной или несколькими территориями. Маршрутизация поездов включает инструкции для автоматического перевода поезда с пути на путь, установку ограничения скорости, подачу и отправку автомобилей, разрешение и запрещение движения поезда в зависимости от занятости определенных участков пути. Диспетчеры могут отмечать состояние путей впереди по маршруту поезда для сообщения его инженеру поезда. Поезда могут быть остановлены системой управления железной дорогой (вручную или автоматически), когда обнаруживается опасность (такая, как выход поезда из графика, поврежденные пути, возможность столкновения). Диспетчеры могут также вызвать любую информацию, доступную инженерам отдельных поездов, послать распоряжения по движению, установить параметры околопутевых устройств, пересмотреть план.

Расположение путей и околопутевое оборудование могут со временем изменяться. Число поездов и маршруты их движения могут изменяться ежедневно. Система должна обеспечивать возможность подключения новых датчиков, сетей передачи данных, оборудования, выполненного по более совершенным технологиям.

Мы выбрали Ada, потому что из всех языков, используемых нами в этой книге, он наиболее пригоден для решения больших задач. Действительно, Ada был выбран для многих крупных проектов, таких, как Расширенная автоматическая система для FAA, Европейская космическая станция и системы управления финансами Американской армии и крупнейшими банками в Финляндии.

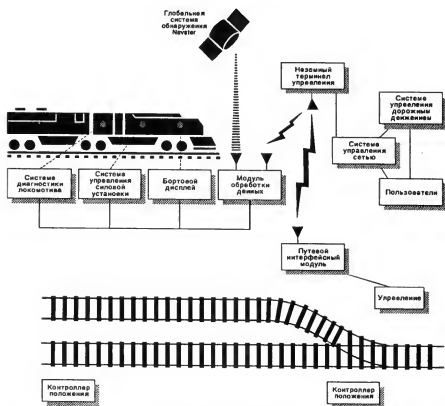


Рис. 12-1. Диаграмма основных элементов системы управления движением.

12.1. АНАЛИЗ

Определение границ задачи

Для большинства людей, живущих в США, поезда являются продуктом давно ушедшей эпохи; в Европе ситуация совершенно противоположная. В отличие от США в Европе мало национальных и международных автомобильных магистралей, а цены на бензин и автомобили сравнительно высоки. Таким образом, поезда составляют основу транспортной сети континента; по десяткам тысяч железнодорожных путей перевозят ежедневно людей и грузы и в городах, и между странами. Ради справедливости отметим, что в США поезда играют по-прежнему важную роль в перевозке грузов. Добавим, что с увеличением городов их центры становятся все более и более перегруженными, и легкий рельсовый транспорт становится главным претендентом для решения проблем борьбы с перегрузками и загрязнения окружающей среды двигателями внутреннего сгорания.

До сих пор железные дороги являются коммерческими и, следовательно, они должны быть прибыльными. Железнодорожные компании обязаны

постоянно поддерживать баланс между требованиями экономики и безопасности, с одной стороны, и нарастающей интенсивностью перевозок, с другой, что противоречит их эффективности и правилам составления расписаний. Эти противоречия наводят на мысль, что решения об управлении движением поездов необходимо получать автоматически, в том числе и контроль за всеми элементами железной дороги с помощью компьютера. Такие автоматические и полуавтоматические системы сегодня существуют в Швеции, Великобритании, Германии, Франции и Японии [12]. Подобная система, называемая Расширенной системой управления железнодорожным транспортом, существует в Канаде и США. Ею пользуются следующие компании: Amtrak, Burlington, the Canadian National Railway Company, CP Rail, CSX Transportation, the Norfolk and Western Railway Company, the Southern Railway Company, Union Pacific [3]. Эффект от каждой из этих систем и экономический, и социальный; результатом их внедрения являются: низкая операционная стоимость, более эффективное использование ресурсов, а также увеличение безопасности (как побочный продукт). Выше сформулированы требования на высоком уровне к системе управления движением поездов. Очевидно, эти требования сильно упрощены. На практике требования к системе настолько обширны, что к ним обращаются только после демонстрации жизнеспособности автоматически принимаемых решений; потом после многих сотен человеко-месяцев анализа в работу вовлекаются множество экспертов в данной области, и только после этого привлекаются пользователи и клиенты системы. В конечном итоге требования на большую систему могут состоять из тысяч страниц документации, специфицирующей не только базовое поведение системы, но и включающей такие подробные детали, как макеты экранов человеко-машинного интерфейса.

Исходя из этих высокоуровневых требований, мы можем сделать два замечания о процессе разработки системы управления движением:

- * Процесс проектирования должен быть открыт для развития.
- * Реализация должна по возможности больше опираться на существующие практические стандарты.

Наш опыт разработки больших систем подсказывает, что первоначальная формулировка требований никогда не бывает полной, она всегда в некоторой степени неопределенна и противоречива. Мы не должны забывать, что процессу проектирования свойственна некоторая неопределенность, и, следовательно, мы должны быть готовы к корректировкам проекта. Эти корректировки могут быть или просто добавлением новых деталей, или постепенным интерактивным процессом. Мы уже говорили в гл. 4, что такое противоречие дает и пользователям, и разработчикам лучшее понимание предмета проектирования, чем попытка написать требования на систему без предварительного прототипирования. Кроме того, необходимо учитывать, что на создание большой системы может быть затрачено несколько лет. За это время уйдет вперед технология аппаратной части системы. Это повлечет за собой изменения в аппаратно-независимых модулях программной части. Мы считаем, что для уменьшения количества изменений проектировщики должны максимально использовать существующие стандарты на связь, сети передачи данных, машинную графику и датчики. Разработка же новых нестандартных аппаратных и программных средств приведет к повышению вероятности неудачи, которая для большинства систем и без того высока. Нашей же целью является снизить эту вероятность до минимума. Мы не затрагиваем вопросы анализа больших систем, так как книга в основном посвящена проектированию.

ванию. Да и вряд ли можно рассмотреть все вопросы анализа в одной книге. Но мы хотим отметить, что объектно-ориентированный анализ — это идеальный метод анализа для систем, подобных нашей. И прежде всего потому, что он позволяет разговаривать разработчикам и пользователям на одном языке на основе общего словаря проблемной области. В результате объектно-ориентированного анализа должны быть определены ключевые абстракции и предполагаемое поведение системы.

В отличие от предыдущих глав размеры данной задачи не позволяют привести здесь детальный проект и тем более полную реализацию системы. Поэтому мы покажем лишь процесс проектирования наиболее высокого уровня системы, записанный с помощью объектно-ориентированной нотации.

Требования к системе и к программному обеспечению

Крупные проекты обычно вырастают из более мелких, которые создавались единым коллективом. Как правило, архитектура системы тоже создается этим коллективом, а потом части системы реализуются различными организациями или различными коллективами внутри одной организации. На стадии анализа системы происходит создание модели состояний из аппаратной и программной частей. Многие считают, что это уже не анализ, а проектирование. Это спорный вопрос. В самом деле, трудно понять, что показано на диаграмме на рис. 12-1, требования или проект системы. Несмотря на это, хорошо видно, что на этой стадии разработки архитектура системы является принципиально объектно-ориентированной. Например: в ней присутствуют такие сложные объекты, как системы управления двигателем и система управления железной дорогой. Обе они выполняют основные функции системы. Это как раз то, о чем мы говорили в гл. 4 — объекты самого высокого уровня абстракции группируются по функциональным признакам. Поэтому процесс анализа в данном случае мало отличается от процесса проектирования.

Когда мы имеем архитектуру на уровне блок-диаграмм (рис. 12-1), мы должны разработать системные требования на уровне каждого блока, как это сделано выше (разд. «Требования к системе управления движением»). Для большей детализации мы можем использовать диаграммы потоков данных, объектов или структурированный текст, чтобы проиллюстрировать взаимодействие различных частей системы, в том числе влияние уровня детализации на поведение системы в целом.

Очевидно, мы должны преобразовать требования к системе в требования к программной и аппаратной частям системы, так чтобы различные организации могли одновременно заниматься отдельными частями задачи. Совместное создание аппаратного и программного обеспечения — сложная задача, особенно если эти части связаны достаточно слабо и создаются разными фирмами. Иногда интуитивно ясно, какая аппаратура будет использоваться. Например, можно использовать готовые терминалы или рабочие станции на бортовых дисплейных системах и для отображения информации в центрах управления железной дорогой. Очевидно также, что программы могут хорошо составлять расписания поездов. Окончательное решение о том, какую основу — аппаратную или программную — использовать в каждом конкретном случае, зависит как от привычек разработчиков, так и от многих других факторов. Специализированную аппаратуру можно использовать, когда важнее производительность, а программы, когда более важна гибкость.

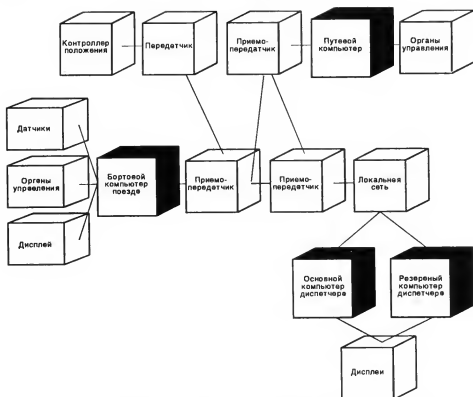


Рис. 12-2. Диаграмма процессов для системы управления движением.

При решении нашей задачи мы считаем, что первоначальный вариант аппаратной архитектуры следует из архитектуры системы. Это предположение не является окончательным, оно дает нам отправную точку для уточнения требований к программному обеспечению. Действительно, чтобы продолжить проектирование, нам необходима свобода выбора аппаратных и программных средств обеспечения: позже станет ясно, какая дополнительная аппаратура нам необходима, или что предпочтительнее для выполнения определенных функций: программы или аппаратура.

На рис. 12-2 показано целевое аппаратное обеспечение для системы управления движением; здесь используются наши обозначения для диаграмм процессов. Эта архитектура процессов соответствует блок-диаграмме системы на рис. 12-1. В частности, существуют один бортовой компьютер на каждом поезде, охватывающая локомотив система сбора и отображения информации на локомотиве, система управления двигателем, бортовая дисплейная система и устройство управления данными. Мы предполагаем, что некоторые бортовые устройства, такие, как дисплей, обладают интеллектом, и считаем, что они не нуждаются в программировании. Мы полагаем, что каждый локаль-

ный транспондер подсоединен к передатчику, который может посылать сообщения на проходящий мимо него поезд; компьютер к локальному транспондеру не подсоединен. Все группы околупутевых устройств (каждое из которых входит в устройство околупутевого интерфейса) управляются компьютером, который может взаимодействовать с проходящим поездом или с контроллером наземного терминала через их передатчики и приемники. Каждый контроллер наземного терминала присоединяется через сеть передачи данных к диспетчерскому центру (который входит в систему управления железной дорогой). Для обеспечения бесперебойного обслуживания мы решили разместить на каждом диспетчерском центре два компьютера: основной и резервный (который подключится в случае отказа основного компьютера). В период простоя резервный компьютер может использоваться для обслуживания других, низкоприоритетных пользователей.

Сданная в эксплуатацию система управления движением может содержать сотни компьютеров: по одному на каждый поезд, по одному на каждое околупутевое интерфейсное устройство, и по два на каждый диспетчерский центр. На диаграмме процесса показано только несколько из этих компьютеров, так как излишне показывать повторяющуюся конфигурацию.

Как мы уже говорили в гл. 7, здравый смысл подсказывает, что при разработке большого проекта огромную роль играют разумность и ясность интерфейсов между ключевыми частями системы. Особенно это важно для интерфейса между программной и аппаратной частями системы. В начале работы над проектом интерфейс может быть определен не полностью, но он должен быть достаточно быстро формализован, чтобы различные части системы можно было разрабатывать, тестировать и выпускать одновременно. Хорошо определенный интерфейс позволяет производить сборку системы без существенных переделок ее частей. Кроме того, мы не рассчитываем, что все разработчики, участвующие в проекте, будут одинаково сильны в программировании. Спецификации ключевых абстракций и механизмов позволят всем воспользоваться результатами работы сильнейших системных архитекторов.

Ключевые абстракции и механизмы

В результате изучения требований к системе управления движением становится видно, что реально мы должны решить четыре подзадачи:

- Сеть передачи данных.
- База данных.
- Человеко-машинный интерфейс.
- Аналоговые устройства управления в реальном времени.

Эти четыре проекта представляют наибольший риск при разработке проекта нашей системы.

Действительно, нить, которая связывает всю систему воедино, — это распределенная сеть передачи данных. Сообщения с помощью радио передаются с транспондеров на поезд, между контроллерами наземных терминалов, между поездами и околупутевыми интерфейсными устройствами, между контроллерами наземных терминалов и околупутевыми интерфейсными устройствами. Кроме того, сообщение должно передаваться между диспетчерскими центрами и отдельными контроллерами наземных терминалов. Надежная работа всей системы обеспечивается своевременными и надежными приемом и передачей сообщений. Кроме того, эта система должна одновременно хранить местоположения и маршруты множества поездов. Мы должны хранить

постоянно обновляемую информацию даже в случае попыток одновременно записать и считать информацию из сети передачи данных. Это фундаментальная проблема распределенных баз данных.

Проектирование человеко-машинного интерфейса требует решения особого множества задач. Дело в том, что пользователями системы в основном являются инженеры поездов и диспетчеры и лишь немногие из них умеют квалифицированно использовать компьютер. Пользовательский интерфейс операционных систем, таких как Unix, пригоден для специалиста-программиста, но слишком неудобен для пользователей готового программного продукта, такого, как система управления движением. Следовательно, все формы взаимодействия должны быть спроектированы в расчете на эту особую группу пользователей. Наконец, система управления движением должна взаимодействовать с разнообразными датчиками и исполнительными механизмами. Остатываясь здесь на природе этих устройств, отметим, что контроль и управление ими сходны и будут осуществляться одним и тем же способом.

Каждую из этих четырех задач можно решать по отдельности. Системные архитекторы должны разработать ключевые абстракции и механизмы, общие для каждой задачи, и тогда мы можем выбрать экспертов для решения каждой отдельной подзадачи одновременно с остальными. Из краткого проблемного анализа этих четырех задач мы находим, что существуют три высокоуровневые ключевые абстракции:

- * Поезда Поезда и автомобили.
- * Пути Профиль пути впереди поезда, качество и околопутевые устройства.
- * Планы Составление, упорядочивание, устранение накладок, назначение полномочий и выработка команд.

Каждый поезд характеризуется текущим положением и может иметь только один активный план движения, который описывает прохождение поезда по его маршруту. Мы можем выделить ключевой механизм для каждой из четырех подзадач:

- * Механизм передачи сообщений.
- * Механизм планирования движения поездов.
- * Механизм отображения информации.
- * Механизм датчиков.

Почему мы выделяем эти механизмы еще до начала процесса проектирования? Дело в том, что эти четыре механизма — основа нашего проекта. Они являются наиболее сложными и рискованными частями системы. Важно, чтобы мы выбрали лучшее решение из всех возможных. Ведь именно эти проектные решения определяют дальнейшую работу всего коллектива.

12.2. ПРОЕКТИРОВАНИЕ

Механизм передачи сообщений

При анализе механизма *передачи* сообщений мы обращаемся к концепции словаря предметной области, как к наиболее высокоуровневой абстракции. Например, типичное сообщение в системе управления движением состоит из сигнала запуска околопутевых устройств, признаков прохождения по-

езда через определенный участок пути и приказов диспетчера для инженера поезда. Все эти виды сообщений могут передаваться внутри системы управления движением на двух уровнях:

- Между компьютерами и устройствами.
- Между компьютерами.

Сейчас нас интересует второй уровень передачи сообщений. Так как наша система включает территориально распределенную сеть передачи данных, мы должны учесть такие факторы, как помехи, отказы оборудования и секретность передачи информации.

Анализ требований к связи. Первым шагом при определении сообщений в системе может быть анализ взаимодействия каждой пары связанных между собой компьютеров (рис. 12-2). Для каждой пары компьютеров мы должны ответить на два вопроса: какая информация будет передаваться с компьютера на компьютер? Какой уровень абстракции будет иметь эта информация? Мы должны при этом использовать последовательный итеративный подход, пока не будем уверены, что передаются правильные сообщения и в системе связи нет «узких» мест («узкие» места могут возникать из-за перегрузки линий связи либо в том случае, если длина сообщений или слишком мала, или слишком велика).

Очень важно, чтобы на данном этапе проектирования внимание было сосредоточено на сути, а не на форме сообщений. Часто системные архитекторы начинают проектирование с выбора битового представления сообщений. Когда в реальной задаче делается преждевременный выбор, такой, как низкоеуровневое представление, это обязательно приведет к изменениям в дальнейшем и, таким образом, затронет всех, кто пользовался этим представлением. Кроме того, на этой стадии проектирования у нас пока нет полной информации, как будут использоваться данные сообщения, и, следовательно, мы не можем судить, какое представление будет оптимальным по размеру и времени передачи.

Концентрируя внимание на сути сообщений, мы рассмотрим все классы сообщений. Другими словами, мы должны определить назначение и смысл каждого сообщения, а так же операции для их содержательной обработки.

На диаграмме классов (рис. 12-3) показаны некоторые наиболее важные сообщения в системе управления движением. Заметим, что все сообщения в конечном итоге являются экземплярами класса Message, который обобщает поведение для всех классов. Три абстрактных класса представляют главные категории сообщений: сообщение о состоянии поезда, сообщение о плане движения поезда, сообщение олопотуевого устройства. Каждый из этих трех классов в дальнейшем будет специализирован. В результате проектирования должны появиться сотни таких специализированных классов. Таким образом, существование обобщающих абстрактных классов чрезвычайно важно; без них мы получили бы сотни несвязанных и, следовательно, сложных при использовании модулей, каждый из которых реализует специализированный класс. По мере проектирования мы будем создавать другие важные группы сообщений и затем придумывать для них другие специализированные промежуточные абстрактные классы.

Реализация проекта. Как мы реализуем этот проект на языке Ada, который является объектным языком программирования и, таким образом, не имеет прямой поддержки механизма наследования? На практике мы проводим проектирование так, как будто наследование возможно, а затем, если

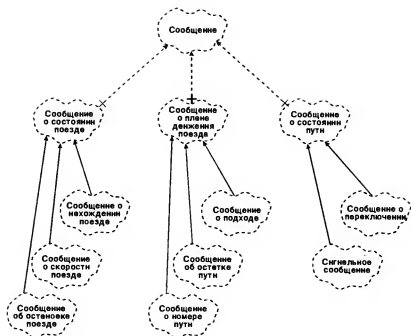


Рис. 12-3. Диаграмма классов сообщений.

язык не имеет прямой поддержки механизма наследования, эмулируем его. В Ada общий подход заключается в использовании комбинации параметризованных пакетов (представляющих параметризованные классы) и различных собственных типов (представляющих абстрактные классы). Из рис. 12-3 видно, что класс *Message* является параметризованным классом, который реализуется в Ada как тип, экспортируемый из параметризованного пакета, и содержит операции, общие для всех сообщений, такие, как передача и прием. Затем абстрактные классы, подобные *Train_Status_Message*, могут быть представлены как собственные типы. Например, мы можем написать класс *Train_Status_Message* следующим образом:

```

with System_Classes;
package Train_Status_Messages is

  type Kind is (Train_Location, Train_Speed, Train_Stop);
  type Train_Status_Message (The_Kind : Kind) is private;

  procedure Set_Location (The_Message : in out Train_Status_Message;
                          The_Location : in System_Classes.Location);
  procedure Set_Speed (The_Message : in out Train_Status_Message;
                       The_Speed : in System_Classes.Speed);
  procedure Stop (The_Message : in out Train_Status_Message);

```



Рис. 12-4. Механизм передачи сообщений.

```

function Location_Of (The_Message : in Train_Status_Message)
  return System_Classes.Location;
function Speed_Of (The_Message : in Train_Status_Message)
  return System_Classes.Speed;
function Is_Stopped (The_Message : in Train_Status_Message) return Boolean;

```

private

```

...
end Train_Status_Messages;

```

Пакет `System_Classes` содержит общие определения для таких классов, как `Location` и `Speed`.

Как показано на диаграмме классов (рис 12-3), каждый экземпляр класса `Message` добавляет новый тип. Так как эти экземпляры структурно связаны между собой, то подклассы промежуточных классов (подобных `Train_Status_Message`) добавляют совместные друг с другом типы.

Когда наш интерфейс будет содержать все наиболее важные сообщения, мы можем начать писать программы, основанные на этих классах для моделирования протоколов передачи сообщений. Эти программы можно использовать для тестирования различных частей системы.

Диаграмма классов на рис. 12-3, конечно, неполна. Ясно, что в первую очередь необходимо разрабатывать наиболее важные сообщения, а все остальные добавлять по мере того, как будем обнаруживать менее общие формы взаимодействия. Использование объектно-ориентированного проектирования позволит добавлять эти сообщения без нарушения существующих частей системы, так как эти изменения задуманы нами с самого начала.

Если мы удовлетворены структурой классов, мы можем начать проектирование самого механизма передачи сообщений. Здесь возникают две противоречащие друг другу цели: придумать механизм для надежной доставки сообщений и сделать это на достаточно высоком уровне абстракции, так чтобы клиенту не надо было заботиться о доставке сообщения. Такой механизм передачи сообщений позволит клиентам иметь упрощенное представление о процессе передачи.

На рис. 12-4 показан результат проектирования механизма передачи сообщений. Для отправки сообщения применяют операцию `Send` (определенную в подклассе `Message`) к объекту сообщения. Эта операция по очереди вызывает асинхронную операцию `Send`, применяемую к некоторому сосредоточенному объекту `Message_Manager`. Мы сделали эту операцию асинхронной, чтобы клиент не ждал, пока сообщение будет отправлено по радио, что тре-

бует времени для кодирования, декодирования и повторных передач из-за помех. В конечном итоге объект `Message_Manager` преобразует значение каждого специализированного объекта сообщения в некоторую каноническую форму, вероятно содержащую поток букв и цифр, а затем направляет этот поток в соответствующее место через сеть передачи данных.

При проектировании объекта `Message_Manager` мы помещаем его на прикладной уровень OSI модели, принятой в ISO [4]. Это позволит всем клиентам-передатчикам и клиентам-приемникам работать на самом высоком уровне абстракции и общаться друг с другом в собственных терминах. Назначение объекта `Message_Manager` состоит в том, чтобы скрыть все низкоуровневые детали физической передачи этих сообщений.

Рассмотрим внутреннее представление объекта `Message_Manager`. Этот объект должен находиться на каждом компьютере, который отправляет или получает сообщения. Его основная семантика гарантирует передачу и прием всех видов сообщений. Так как передача может быть прервана из-за возникновения ошибок (таких, как ошибки при радиопередаче), то эти объекты должны реагировать на подобные исключительные ситуации.

Параметризованные пакеты Ada хорошо приспособлены для реализации таких объектов. Сначала мы параметризуем пакет дискретным типом, представляющим все возможные типы сообщений. Этот параметризованный пакет скрывает устройство двух внутренних параметризованных процедур: одну для отправки и другую для приема сообщений. Реализация классов сообщений может содержать экземпляр каждой или обеих процедур. Эти процедуры включают операцию по преобразованию каждого типа сообщений в каноническую форму. Таким образом, каждый класс сообщений скрывает детали низкоуровневого представления этого сообщения. Все классы сообщений могут быть построены на основе набора стандартных функций для преобразования сообщений.

Мы можем описать пакет `Message_Manager` следующим образом:

```
with Network_Classes;
generic
  type Message_Kind is (< >);
package Message_Manager is

  generic
    type Message is private;
    with function String_To_Message (The_String : in String)
      return Message;
  procedure Send
    (The_Message : in Message;
     From         : in Network_Classes.Id;
     To           : in Network_Classes.Id;
     Retry        : in Network_Classes.Retry_Count;
     Acknowledge  : in Boolean);

  generic
    type Message is private;
    with function Message_To_String (The_Message : in Message) return String;
  procedure Receive
    (The_Message : out Message;
     From         : in Network_Classes.Id;
     To           : in Network_Classes.Id);

  function Kind_Of_Message return Message_Kind;

  Transmission_Error : exception;
  Reception_Error    : exception;
end Message_Manager;
```

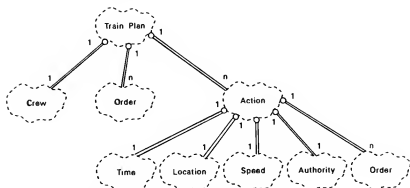


Рис. 12-5. Диаграмма классов для плана поезда.

Пакет `Network_Classes` содержит базовые определения, такие, например, как `ID` — тип для уникальной идентификации узлов в сети передачи данных. Обратите внимание на описание двух исключений: `Transmission_Error` и `Reception_Error`, которые могут быть вызваны тогда, когда сообщение не может быть нормально передано или принято.

Мы считаем, что окончательная реализация этого параметризованного пакета будет немного сложнее. Например, мы могли бы добавить к операциям передачи и приема опции для синхронизации связи или для тайма-аутов. Мы также можем добавить больше исключений, чтобы точнее идентифицировать причины ошибок передачи и приема. В любом случае спроектированная таким образом модель позволяет всем клиентам работать на самом высоком уровне абстракции, взаимодействуя друг с другом в терминах их специфической области приложения. С другой стороны, эти модули скрывают множество сложных вещей. Отметим, для того чтобы иметь дело с физической передачей сообщений, эти модули должны быть построены на основе общей низкоуровневой абстракции. Это позволяет выполнять шифрование и дешифрование и использовать коды для обнаружения и исправления ошибок, что в свою очередь обеспечивает надежную связь в случае ошибок передачи или отказов оборудования.

Механизм планирования движения поездов

Выше мы говорили, что концепция планирования движения поездов является центральной для функционирования системы управления движением. Каждый поезд имеет один активный план, а каждый план назначается только одному поезду. Он может содержать много различных приказов и точек на пути. Наш первый шаг состоит в определении, из каких частей состоит план поезда. Для этого мы должны определить всех потенциальных клиентов плана и их способ использования этого плана. Например, некоторым клиентам может быть разрешено составлять планы, другим может быть разрешено корректировать планы, а остальные смогут только читать планы. В этом смысле план действует как хранилище информации, связанной с маршрутом одного отдельного поезда и действиями, которые надо произвести во время движения. Примером таких действий может быть подача или отправка автомобилей.

Время	Место	Скорость	Автор	Порядок
0800	Pueblo	Как почтой		Покинуть сортировочную станцию
0815	Pueblo	Как почтой	See Yardmaster	Подать 50 машин
1100	Colorado Springs	40 MPH		Отправить 30 машин
1300	Denver	45 MPH		Отправить 20 машин
1600	Pueblo	Как почтой		Вернуться на сортировочную станцию

На рис. 12-5 приведены проектные решения, касающиеся структуры класса Train_Plan. Как и в гл. 10, мы используем диаграмму классов, чтобы показать части, из которых состоит план движения поезда (подобно тому, как это делается на традиционных диаграммах отношений объектов). Мы видим, что каждый план содержит один поезд и может содержать много приказов и действий. Мы ожидаем, что эти действия будут упорядочены во времени и с каждым действием связана такая информация, как время, местоположение, скорость, полномочия, приказы. Например, план может содержать следующие действия:

Так же как для класса Message и его подклассов, мы можем в первую очередь спроектировать наиболее важные элементы плана поезда; его детали будут создаваться по мере того, как мы будем применять план к новым задачам.

Тот факт, что мы можем иметь множество активных и неактивных планов поездов одновременно, ставит вопрос о создании базы данных, о которой мы уже говорили. Диаграмма классов на рис. 12-5 может служить наброском логической схемы этой базы данных. Следующий вопрос, который возникает при этом состоит в следующем: где находится план поезда?

В идеальном случае без помех или задержек при передаче и неограниченных ресурсах компьютера лучше всего было бы разместить все планы поезда в единой централизованной базе данных. Такой подход обеспечивает существование единственного экземпляра каждого плана поезда. Однако реальные условия делают это решение не эффективным. Мы предполагаем, что будут задержки при передаче и производительность процессора ограничена. Таким образом, скорость доступа к плану, который расположен в диспетчерском центре, с поезда не будет отвечать всем требованиям реального времени. Однако мы можем создать иллюзию наличия централизованной базы данных с помощью нашего программного обеспечения. Наше решение заключается в том, что планы поездов будут располагаться на компьютерах диспетчерского центра, а копии индивидуальных планов — распределяться при необходимости по сети передачи данных. Для обеспечения необходимой эффективности компьютер каждого поезда может хранить копию активного плана. Таким образом, бортовое программное обеспечение может сделать запрос по этому плану с минимальной задержкой. Если план изменяется в



Рис. 12-6. Механизм планирования движения поезда.

результате действий диспетчера или (что менее вероятно) по решению инженера поезда, наши программы должны гарантировать, что все копии этого плана обновятся одинаково.

На рис. 12-6 показано, как происходит передача и обновление копий плана. Первоначально единственная копия плана движения находится в централизованной базе данных на диспетчерском центре. Затем по сети может быть разослано любое число копий этого плана. Когда копия плана посылается клиенту, в базе данных записывается ее местоположение. Теперь предположим, что в результате действий инженера поезда появилась необходимость изменить план движения этого поезда. Сначала изменяется копия плана, находящаяся на поезде. Затем сообщение об изменениях посылается в централизованную базу данных на диспетчерский центр. После того как план изменился в базе данных, сообщения об изменениях рассылаются всем остальным клиентам, которые имеют у себя копии данного плана.

Этот механизм работает и тогда, когда изменения в план движения вносит диспетчер, в этом случае сначала изменяется копия плана в базе данных, а затем сообщения об изменениях рассылаются по сети остальным клиентам. Как в обоих случаях осуществляется передача изменений? Для этого мы используем механизм передачи сообщений, разработанный нами ранее. Заметим, что в результате проектирования мы добавили новое сообщение о плане движения поезда. Работа с этим сообщением базирруется на уже существующем низкоуровневом механизме передачи сообщений.

Использование существующей коммерческой системы управления базами данных позволит обеспечить резервирование, отходы назад, ведение контрольного журнала и секретность информации.

Механизм отображения информации

Использование готовых технологических решений для механизма отображения информации, так же как для базы данных, позволит нам сосредото-

читься на отдельных частях нашей задачи. Этого можно добиться, если использовать графические стандарты, такие, как GKS, PHIGS или XWindows. Использование готовых графических программных средств поднимает уровень абстракции нашей системы настолько, что разработчикам не надо беспокоиться об обработке отображенной информации на уровне пикселей.

Рассмотрим, например, отображение информации о профиле и качестве участков пути. Согласно требованиям отображение может появиться в двух местах: на диспетчерском центре и на поезде (где отображается путь только впереди по маршруту движения поезда). Считая, что мы имеем некоторый класс, экземпляры которого представляют участки пути, можно рассмотреть два подхода к визуализации этого объекта. В соответствии с первым подходом, создается специальный объект, управляющий отображением, который преобразует состояние объекта в визуальную форму. По второму мы отказываемся от этого специального внешнего объекта, но вместо этого в каждом нашем объекте помещаем информацию, как отображать самого себя. Мы считаем, что второй подход предпочтительней, так как он проще и лучше отражает сущность объектной модели, хотя и не лишен недостатков.

В конце концов у нас будет множество разновидностей отображенных объектов, каждый из которых создан разными группами разработчиков. Если реализовывать каждый отображаемый объект отдельно, то получим избыточный код, разный стиль и большую путаницу. Правильнее проанализировать все разновидности отображаемых объектов, определить, какие из них общие элементы, и создать множество промежуточных классов, которые обеспечат отображение этих общих элементов. В свою очередь эти промежуточные классы могут быть построены на основе существующих низкоуровневых графических пакетов.

На рис. 12-7 показана реализация всех отображаемых объектов, разделяющих общие промежуточные классы. Эти классы построены на основе низкоуровневых услуг пакета XWindows, которые скрыты от всех производных классов. Основное достоинство этого подхода заключается в том, что уменьшается влияние любых изменений аппаратуры или программ на низком уровне. Например, если нам надо заменить наши дисплеи на более или менее мощные устройства, придется изменять программы только в Train_Display_Uilities.

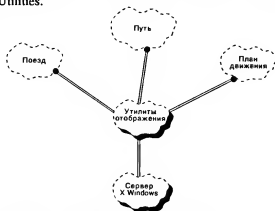


Рис. 12-7. Механизм отображения информации.

Без такой декомпозиции программ нам бы пришлось вносить изменения в каждый отображаемый объект при любых изменениях на нижнем уровне.

Механизм датчиков

Выше мы говорили, что система управления движением должна включать в себя большое количество разнообразных датчиков. Например, на каждом поезде датчики следят за температурой масла, количеством топлива, дроссельной установкой, температурой воды, нагрузкой на тяговый стержень и т.д. Активные датчики околопутевых устройств сообщают текущее положение своих переключателей. Все виды возвращаемых датчиками величин разные, но обработка этих величин производится похоже. Например, допустим, что наш компьютер использует привязанный к фиксированным адресам памяти ввод-вывод, тогда данные каждого датчика одинаково читаются из определенной области памяти и только потом интерпретируются зависящим от конкретного датчика способом. Так же большинство датчиков должно периодически проверяться. Для всех датчиков справедливо, что, если величины находятся в пределах заданных границ, клиенту не сообщается ничего, кроме поступления новой величины. Если же величина вышла из заданных границ, об этом могут быть оповещены разные клиенты. Если такая величина выйдет далеко за границы, мы можем дать какой-то сигнал тревоги и побудить других клиентов предпринять решительные действия (например, когда давление масла на локомотиве поднимается до опасного уровня).

Повторение описанных выше действий не только приводит к монотонности и появлению ошибок, но и к созданию избыточного кода. Если мы с самого начала не выделим общие для всех датчиков части, то все разработчики предложат свои решения одной и той же задачи. Это в свою очередь приведет к сложностям при сопровождении системы. Поэтому для выявления общих свойств необходимо провести анализ всех дискретных и циклических датчиков.

Рассмотрим внешнее представление этого базового класса и ответим на вопрос: какие действия мы можем совершать над их экземплярами? Здесь возможны три варианта: два модификатора и один селектор. Во-первых, мы должны показать, как провести начальную установку параметров датчика. Затем, мы должны сказать, как часто обновляются величины датчиков, и определить границы его допустимых и недопустимых значений. Для полноты мы должны также предусмотреть операцию, производящую остановку датчика. И наконец, мы можем добавить селектор для принудительного чтения текущего значения датчика.

Возможны ли во время работы датчика аномальные ситуации? Клиент может пытаться запустить уже работающий датчик (мы не рассматриваем попытку остановить неработающий датчик как ошибку). Кроме того, может произойти неисправимая ошибка, которая связана с аппаратурой и которую мы можем обнаружить при попытке прочесть его значение. Средствами обработки особых ситуаций в языке Ada легко сделать так, чтобы эти ошибки не влияли на пользователей.

Операции, которые мы можем выполнить над объектом, это — только половина устройства объекта; мы также должны учитывать, что операции этого объекта используют другие объекты. Для этого базового класса датчиков мы можем рассмотреть четыре операции: три модификатора и один селектор. Три модификатора соответственно обеспечивают получение значения

датчика, предупреждение клиента о ненормальном значении и подачу сигнала тревоги клиенту, если значение достигает опасной величины. Нам необходим селектор, который сообщает датчику, как преобразовать входной поток битов в величину, соответствующую данному датчику. Эти данные лучше связывать с классами датчиков.

В Ada недопустимы переменные в подпрограммах, но допустимы параметризованные блоки, содержащие параметры подпрограмм. Это решение удовлетворяет нашим требованиям, так как физический датчик размещается статически. Мы можем записать результат наших проектных решений на Ada следующим образом:

```
with System_Classes;
generic
  type Value is private;
  with procedure Notify_Client (The_Value : in Value);
  with procedure Warn_Client   (The_Value : in Value);
  with procedure Alert_Client  (The_Value : in Value);
  with function Scale          (The_Raw_Value : in Integer) return Value;
package Periodic_Sensor_Manager is

  type Sensor (Memory_Address : System_Classes.Address) is private;

  procedure Start (The_Sensor      : in      out      Sensor;
                  Frequency        : in      Duration;
                  Warning_High_Limit : in      Value;
                  Warning_Low_Limit : in      Value;
                  Alert_High_Limit  : in      Value;
                  Alert_Low_Limit   : in      Value);
  procedure Stop (The_Sensor      : in      out      Sensor);

  function Current_Value_Of (The_Sensor : in Sensor) return Value;

  Sensor_Is_Already_Started : exception;
  Sensor_Failure            : exception;

private
  ...
end Periodic_Sensor_Manager;
```

Мы предполагаем, что пакет `System_Classes` содержит соответствующее описание для дискретного типа `Address`, представляющего физическое размещение в памяти. Используя этот тип, мы можем описать тип `Sensor` с различными величинами, представляющими порты ввода-вывода в карте памяти, связанной с этим датчиком.

Так как это параметризованный блок, мы не можем использовать его непосредственно; мы должны обеспечить экземпляр для каждого типа датчика, как это показано на рис. 12-8. Мы ожидаем, что архитекторы системы предоставят ряд стандартных экземпляров, которые другие разработчики будут использовать непосредственно. Придуманый нами механизм работает лучше для циклических аналоговых датчиков и хуже для дискретных датчиков (таких, как переключатели) и для таких, которые регистрируют единичные, а не непрерывные события (такие, как сеанс связи автомобилей с локомотивом). Если наш анализ выявляет много таких датчиков, нам придется разработать сходный механизм и для них.

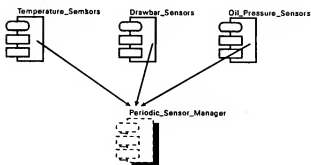


Рис. 12-8. Механизм датчиков.

12.3. РАЗВИТИЕ

Модульная архитектура

Пакеты для больших систем необходимы, но в них недостаточно средств декомпозиции; следовательно, для решения этой проблемы мы должны сосредоточиться на подсистемном уровне декомпозиции. Мы должны разработать модульную архитектуру системы управления движением, представляющую физическую структуру ее программного обеспечения.

Проектирование программного обеспечения очень больших систем часто должно начинаться до полного завершения проектирования аппаратных средств. Оно и занимает больше времени. В любом случае интерфейс должен быть разработан до того, как каждый завершит свои работы. Это означает, что аппаратные и программные средства должны быть изолированы друг от друга настолько, насколько возможно, так чтобы проектирование программных средств можно было осуществлять без привязки к аппаратным средствам. Это означает также, что программное обеспечение должно быть заменяемым. В управляющих и контролирующих системах, таких, как система управления движением, мы должны быть заинтересованы в том, чтобы вводить новые аппаратные средства, которые могут появиться в процессе разработки программного обеспечения.

Мы также должны на ранних этапах разумно провести декомпозицию программного обеспечения системы, чтобы субконтрактные работы над различными частями системы могли проводиться одновременно (особенно если используются различные языки программирования). Как мы уже говорили в гл. 7, существует много нетехнических причин, определяющих физическую декомпозицию больших систем. Наиболее важным в этом случае является вопрос взаимодействия независимых групп разработчиков. Отношения субподрядчиков складываются обычно на достаточно ранних стадиях жизни системы, часто до того, как получена достаточная информация для проведения правильной декомпозиции системы. Мы рекомендуем системным архитекторам провести несколько альтернативных декомпозиций подсистем для того, чтобы быть уверенным в правильности общих решений по физическому проектированию. Можно включить прототипирование в больших масштабах, но без подключения к прототипу реализации подсистем и моделирование за-

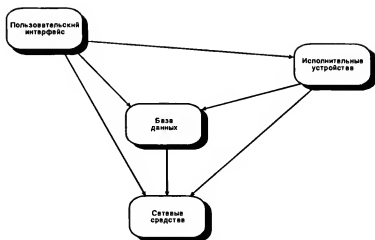


Рис. 12-9. Диаграмма модулей высшего уровня для системы управления движением.



Рис. 12-10. Диаграмма модулей подсистемы Network Facilities.

грузки процессора, маршрутизации сообщений и внешних событий. Прототипирование и моделирование могут послужить основой для нисходящего тестирования после создания системы. Как мы выбирали подходящую декомпозицию подсистемы? В гл. 4 показано, что объекты на высоком уровне обычно группируются в соответствии с их функциональным поведением. Еще раз отметим, что это не противоречит объектной модели, так как в термин «функциональный» мы не вкладываем понятие алгоритмической абстракции, включающей соответствие входа и выхода. Мы говорим, что функциональность системы, которая представляет внешний вид и тестируемое поведение, является результатом совместной работы группы объектов. Таким образом, абстракция высокого уровня и механизмы, о которых мы говорили выше, являются хорошими кандидатами, вокруг которых организуются наши подсистемы. Мы можем сначала допустить существование таких подсистем, а их интерфейс разработать через некоторое время.

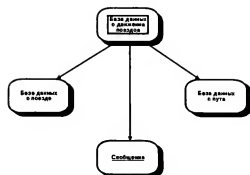


Рис. 12-11. Диаграмма модулей подсистемы Databases.

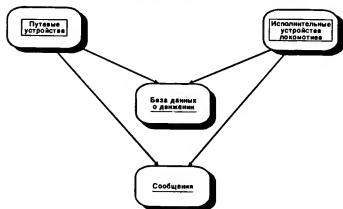


Рис. 12-12. Диаграмма модулей подсистем Devices.

На диаграмме модулей (рис. 12-9) представлены проектные решения по организации верхнего уровня модульной архитектуры системы управления движением. Это высокоуровневая архитектура, каждый уровень которой соответствует функциям четырех подзадач, которые мы выделили выше: сеть передачи данных, база данных, аналоговые устройства управления в реальном времени, человеко-машинный интерфейс.

Спецификация подсистем

Если мы рассмотрим внешнее представление любой из этих подзадач, то найдем, что все они обладают характеристиками объектов. Они уникальны, хоть и статичны, идентифицируемы; это создает большое количество состояний и приводит к очень сложному поведению. Подсистемы используются как хранилища других классов, утилит классов и объектов; таким образом, они лучше всего характеризуются экспортируемыми ресурсами. На практике, при

использовании Ada эти ресурсы представляются в форме логической совокупности компилируемых единиц, многие из которых упакованы.

Диаграмма модулей на рис. 12-9 полезна, но не полна, так как каждая из подсистем на этой диаграмме слишком велика для реализации ее небольшим коллективом разработчиков. Мы должны раскрыть внутреннее представление подсистем верхнего уровня, а затем провести их декомпозицию.

На рис. 12-10 показана диаграмма модулей подсистемы Network Facilities. Мы видим, что одна из подсистем собственная (Radio Communication), а другая экспортируется (Message). Собственный пакет скрывает детали программного управления физическими устройствами, в то время как экспортируемая подсистема обеспечивает спроектированный ранее механизм передачи сообщений.

Подсистема, называемая Databases, построена на основе ресурсов подсистемы Network Facilities и служит для реализации механизма планирования движения поезда, который мы создали выше. Как показано на рис. 12-11, подсистема Databases состоит из нескольких понятных частей. Мы видим, что подсистема Train-Plan Database экспортируется из Database, что позволяет сделать ее ресурсы доступными другим подсистемам, которые явно импортируют их. Подсистема Train-Plan Database также строится на основе ресурсов двух собственных подсистем (Train Database и Track Database) и одной импортируемой подсистемы (Message).

Затем подсистема Devices также естественно декомпозируется на несколько небольших подсистем. Как показано на рис. 12-12, мы решили сгруппировать программы, относящиеся по всем околупутевым устройствам, в одну подсистему, а программы, связанные с исполнительными механизмами и датчиками локомотива, в другую. Эти две подсистемы доступны клиенту подсистемы Devices, и обе построены на основе ресурсов, представляемых Train-Plan Database и Message. Таким образом, мы спроектировали подсистему Devices для реализации механизма датчиков, который мы спроектировали выше.

Как показано на рис. 12-13, мы выбрали декомпозицию подсистемы верхнего уровня User Applications в несколько небольших подсистем, включая Engineer Applications и Dispatcher Applications, для отображения различных долей двух главных пользователей системы управления движением. Подсистема Engineer Applications содержит ресурсы, которые обеспечивают все взаимодействия, специфицированные в требованиях человеко-машинного интерфейса, включая функции анализа системы сбора и отображения информации о состоянии локомотива и системы управления двигателем. Подсистема Dispatcher Applications обеспечивает все функции интерфейса диспетчер-машиниста. Подсистемы Engineer Applications и Dispatcher Applications разделяют общие собственные ресурсы, экспортируемые из подсистемы Dispatcher, которая реализует механизм отображения, описанный нами выше.

В результате проектирования мы получили четыре подсистемы верхнего уровня, окруженные десятью меньшими подсистемами, для которых мы имеем все ключевые абстракции и механизмы, разработанные до этого. Важно, что эти подсистемы формируют блоки для присвоения имен и блоки для создания конфигурации управления и разработки различных версий. Даже на ранних стадиях проектирования мы можем начать создавать множество реализаций, составляющих совместимые версии каждой подсистемы. Как мы говорили в гл. 7, разрабатывать подсистему может один человек, а реализовы-

вать многие. Разработчик детализирует проектирование и реализацию подсистемы и управляет ее интерфейсом с другими подсистемами на том же уровне абстракции. Таким образом, управление разработкой очень больших проектов становится возможным, когда происходит декомпозиция сложных задач.

Основой для выполнения этой работы является инженерное конструирование интерфейсов. После того как интерфейсы сконструированы, они должны тщательно сохраняться. Как мы определяем внешнее представление подсистемы? Для ответа на этот вопрос надо каждую подсистему рассматривать как объект, т.е. мы ставим тот же вопрос, какой мы задавали в гл. 4 для значительно более простых объектов: какие состояния имеет объект, какие действия может выполнить клиент над ним, какие действия он требует от других объектов?

Например, рассмотрим подсистему Train-Plan Database. Она строится на основе трех других подсистем Messages, Train Database, Track Database и имеет нескольких важных клиентов — подсистемы Wayside Devices, Locomotive Devices, Engineer Applications и Dispatcher Applications. Train-Plan Database имеет относительно простые состояния, особенно состояния всех планов поезда. Конечно, ненормально, что эта подсистема должна поддерживать поведение распределенного механизма планирования движением поезда. Итак, с внешней стороны клиент видит монолитную базу данных, но изнутри мы знаем, что на самом деле база данных распределенная, и поэтому должны разместить этот механизм над механизмом передачи сообщений в подсистеме Messages.

Какие действия можно выполнять над Train-Plan Database? Выполнимы все обычные операции базы данных: добавление, удаление, изменение записей и запросы о записях. В конце концов мы соберем все проектные решения в форме пакетов Ada, которые составляют эту подсистему и обеспечивают описание всех ее операций.

На этой стадии проектирования мы продолжаем процесс проектирования для каждой подсистемы. Еще раз отметим, что вероятность того, что эти интерфейсы окажутся правильными с первого раза очень мала. Как и для небольших объектов, опыт подсказывает, что большинство изменений, которые мы произведем в интерфейсах, не затронут лежащие выше уровни, если допустить, что до этого мы правильно охарактеризовали каждую подсистему в объектно-ориентированном стиле.

12.4. ИЗМЕНЕНИЯ

Добавление новых функций

Старое программное обеспечение постоянно сопровождается и дорабатывается, что особенно справедливо для таких больших систем, как наша. Действительно, мы до сих пор находим используемые программы, разработанные двадцать лет назад (которые по программным меркам являются патриархами). Чем больше пользователей применяют систему управления движением и чем лучше мы адаптируем проект к новым реализациям, тем чаще клиенты будут находить новые неожиданные применения для существующих механизмов, создавая необходимость включения в систему новых функций. Рассмотрим единственное добавление к нашим требованиям, называемое обработкой платежной ведомости. Предположим, анализ показал, что в веде-

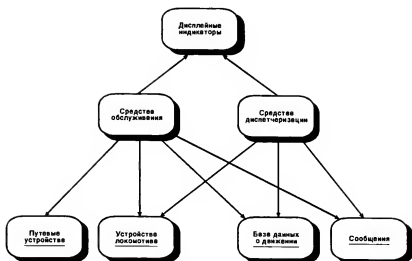


Рис. 12-13. Диаграмма модулей подсистемы User Applications.

нии платежной ведомости железнодорожной компании участвует аппаратура, выпуск которой прекращен, и возник серьезный риск безвозвратной потери платежной ведомости в результате нескольких поломок в системе. В этом случае мы можем объединить обработку платежной ведомости с системой управления движением. Для начала несложно понять, как эти две несвязанные задачи будут решаться; мы можем рассмотреть их как разные приложения, где обработка платежной ведомости работает в фоновом режиме. Дальнейший анализ показывает, что может быть получена большая польза от обработки платежной ведомости. Вспомним, что план поезда содержит информацию о распределении бригад. Отсюда мы можем проследить действительные варианты запланированного распределения бригад и рассчитать затраченное ими рабочее время, время переработки и т.п. Получая эту информацию непосредственно, мы можем обрабатывать платежную ведомость дешевле и быстрее. Что надо сделать для добавления этой функции к нашему уже существующему проекту? Наш подход заключается в том, чтобы можно было добавлять новые подсистемы в подсистему User Applications, представляющую функции обработки платежной ведомости. В данном месте архитектуры такой подсистеме будут видны все важные механизмы, которые служат для ее построения. Это действительно полностью объединенная, хорошо сконструированная объектно-ориентированная система: простые добавления, необходимые для системы, могут быть сделаны достаточно просто построением новых приложений на основе уже существующих механизмов. Предположим, мы хотим ввести более существенное изменение: ввести элементы экспертной системы, создав систему, помогающую диспетчеру, при прокладке маршрутов и предупреждающую об ошибках. Как это требование

отразится на нашем проекте? Незначительно. Мы размещаем новую подсистему между подсистемами Train-Plan Database и Dispatcher Applications, так как база знаний, созданная для экспертной системы, подобна по содержанию Train-Plan Database; кроме того, подсистема Dispatcher Applications является единственным клиентом экспертной системы. Нам предстоит разработать некоторый новый механизм, чтобы доводить рекомендации до конечного пользователя. Например, мы можем использовать архитектуру классной доски так, как мы это делали в гл. 11.

Изменение аппаратных средств

Мы уже говорили, что аппаратные средства развиваются быстрее, чем мы умеем создавать программное обеспечение. Поэтому рабочая аппаратура в больших системах заменяется гораздо раньше, чем программы. Например, после нескольких лет эксплуатации мы можем заменить дисплей на каждом поезде и каждом диспетчерском центре. Как это может повлиять на существующий проект? Если во время разработки проекта мы ограничили интерфейсы подсистем на высоком уровне абстракции, эти изменения аппаратуры приведут к незначительным изменениям программ. Мы изменяем только совокупность программ, относящихся к данным дисплеям, но не для других подсистем, которые не были рассчитаны на особенности данных рабочих станций. Это достигается тем, что поведение всех рабочих станций скрыто в подсистеме Displays. Таким образом, эта подсистема действует как абстрактный пожарный, который защищает остальных клиентов от сложностей разнообразных дисплеев. Таким образом, радикальные изменения в стандартах коммуникации затронут реализацию, но только в ограниченной части. Наш проект гарантирует, что только подсистема, называемая Messages, знает о сетевой коммуникации. Таким образом, фундаментальные изменения в сети не отразятся ни на каком высокоуровневом клиенте; подсистема Messages защищает их от реального мира. Любые изменения, которые мы вводили, не смогли нарушить структуру созданного нами проекта. Это верный признак хорошо спроектированной объектно-ориентированной системы.

Дополнительная литература

Требования к системе управления движением основаны на расширенной системе управления железной дорогой, описанной в работе Murphy [C, 1988]. Трансляция и верификация сообщений появляются фактически во всех командах и управляющих системах. Plinta, Lee и Rissman [C, 1989] предложили проект механизма безопасного распространения сообщений в процессорах распределенных систем. Краткое изложение языка программирования Ada с примерами приведено в приложениях.

Заключение

Книги лишь частично рождаются умом и плотью своих создателей. Большая их часть возникает откуда-то еще, и авторы сидят у своих пишущих машинок в ожидании, когда «случится» книга.

GUY LEFRANCOIS
Of Children

Объектно-ориентированное проектирование претендует на роль последнего слова в методологии проектирования, однако оно представляет собой сплав многих лучших идей в области создания сложных систем. Такие системы уже созданы в различных предметных областях, а сам подход успешно использован как для задач в несколько сот строк кода, так и в 1-10 миллионов строк. Пока нет экспериментальных (практических) доказательств, что данная методология позволит реализовать системы объемом в десятки миллионов строк кода. Можно полагать, что развитие объектно-ориентированного проектирования приведет к тому, что будут созданы системы невообразимой сложности.

Действительно, требования к уровню программных систем резко возрастают. Современные возможности аппаратных средств и общественная потребность в компьютеризации все более и более требуют расширения сферы автоматизации. Потенциальные возможности объектно-ориентированного проектирования, включая процедурные вопросы и документирование, дают свободу энергии человека и позволяют преодолеть препятствия на пути создания сложных систем.

Приложения

Объектные и объектно-ориентированные языки программирования

Методология OOD не накладывает ограничений на использование какого-либо одного языка программирования. В ней может быть применен широкий спектр объектных и объектно-ориентированных языков программирования. Однако нельзя игнорировать особенности кодирования, определяемые языком. «Язык программирования выполняет три функции: 1) инструмент проектирования, 2) средство выражения мыслей программиста, 3) средство формирования инструкций для ЭВМ» [1]. Приложение предназначено для читателей, которые незнакомы с теми языками программирования, которые использованы в книге для описания примеров. Для каждого языка приведены краткое описание и некоторые важные особенности.

П.1. Введение

В настоящее время известно более 2000 различных языков программирования. Это объясняется тем, что каждый язык создавался исходя из конкретных требований определенных предметных областей. Кроме того, новые языки разрабатывались для решения все более и более сложных задач. Опыт применения языков программирования также отражается на представлениях о том, какие элементы языка более существенны или менее необходимы. В развитии языков программирования большое влияние оказали достижения в теории создания компьютерных систем, в частности способы формального описания операторов, модулей абстрактных типов данных и процедур. В гл. 2 говорилось о четырех поколениях языков программирования, которые ориентированы на математические вычисления, алгоритмы, данные и объектно-ориентированные абстракции. Последние достижения в области развития языков программирования связаны с влиянием объектного подхода. Уже насчитывается более 100 различных объектных и объектно-ориентированных языков. Принято называть объектными языки, которые имеют механизмы реализации абстрактных данных и классов; объектно-ориентированными являются те объектные языки, в которых реализован механизм наследования как средство отражения иерархии классов.

Общим предком практически всех используемых объектных и объектно-ориентированных языков является язык Simula, созданный в 1960 г. Далом, Майхраугом и Нейгардом [2]. Язык Simula основывался на идеях языка ALGOL, но был дополнен механизмом наследования и ограничения доступа. Кроме того, Simula — язык, который ориентирован на описание систем и их прототипирование и имеет строгую дисциплину написания программ, отражающую словарь предметной области.

На рис. П-1 показанная генеалогия демонстрирует генеалогию пяти наиболее представительных и распространенных языков программирования объектного и объектно-ориентированного направления: Smalltalk, Object Pascal, C++, CLOS и Ada. Эти языки выбраны для примеров реализации объектного подхода благодаря их значимости в процессе создания сложных систем.

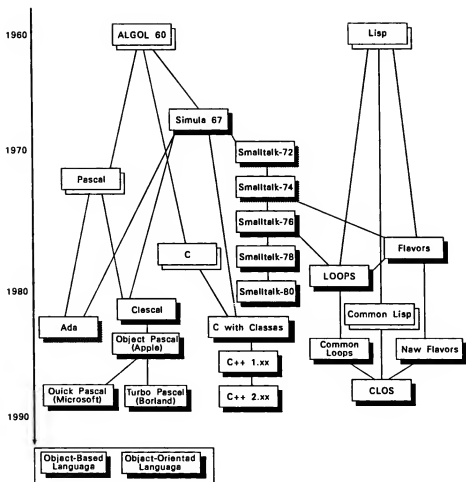


Рис. П-1. Генеалогия объектных и объектно-ориентированных языков программирования.

П.2. ЯЗЫК SMALLTALK

Основные положения

Язык Smalltalk разработан участниками группы исследовательского центра фирмы Xerox в Palo Alto как элемент программы фантастического проекта Dynabook Алана Кая (Alan Kay). В основу языка Smalltalk были положены идеи языка Simula, а также языка FLEX и разработки Сеймора Паперта

и Валласа Фьюрзича (Seymore Papert, Wallace Feurzeig). Smalltalk является одновременно языком программирования и программной оболочкой. Это «чистый» объектно-ориентированный язык, в котором все рассматривается в виде объектов, включая целые числа и классы. Smalltalk, как и язык Simula, является наиболее представительным объектно-ориентированным языком, поскольку он оказал влияние не только на последующие поколения языков программирования, но и на принципы построения графического интерфейса пользователя (например, на наиболее популярные сегодня средства визуализации Macintosh и Motif).

Работа над языком Smalltalk продолжалась почти 10 лет. Главным архитектором проекта Smalltalk на протяжении всей работы был Дэн Ингаллс (Dan Ingalls), но значительный вклад в него внесли также Питер Дейтч, Глен Краснер и Ким Мак-Колл (Peter Deutsch, Glenn Krasner, Kim McCall). Элементы оболочки Smalltalk разрабатывались параллельно Джеймсом Альтхоффом, Робертом Флеганом, Тедом Кехлером, Дианой Мерри и Стивом Рутцом (James Althoff, Robert Flegal, Ted Kaehler, Diana Merry и Steve Rutz). Важную роль в проекте играли также Адель Гольдберг и Дэвид Робсон (Adele Goldberg, David Robson), которые вели учет по ходу проектирования. Известно пять реализаций языка Smalltalk: Smalltalk-72, -74, -76, -78, -80 (цифры обозначают год создания). Реализации Smalltalk-72, -74 заложили основу языка, но не имели механизма наследования.

В последующих версиях был введен общий суперкласс и завершено формирование идеи о том, что все элементы языка должны быть объектами. Smalltalk-80 реализован на многих классах ЭВМ и является сегодня коммерческим продуктом в первую очередь для персональных компьютеров и рабочих станций.

Обзор

По утверждению Ingalls: «Цель проекта Smalltalk состоит в том, чтобы сделать мир информации доступным даже для детей любого возраста. Вся трудность состоит в том, чтобы найти и использовать достаточно простые и эффективные средства, которые позволят отдельному человеку свободно оперировать самой разной информацией от — простых чисел и текста до звуковых и зрительных образов» [4]. Основу языка составляют две простые концепции: 1) все рассматривается в качестве объектов; 2) объекты взаимодействуют путем обмена сообщениями.

В табл. П-1 сведены характеристики языка Smalltalk, имеющие отношения к семи основным элементам объектного подхода. Множественное наследование не отмечено в таблице, но может быть реализовано путем переопределения некоторых простых методов [5].

Пример

Рассмотрим задачу с однородным списком изображений в виде окружности, прямоугольника или закрашенного прямоугольника (аналогично задаче, приведенной в гл. 3). В обширной библиотеке Smalltalk уже определены классы для окружности и прямоугольника, поэтому решение задачи упрощается. Это роль повторного использования компонент. Однако для удобства сравнения предположим, что в нашем распоряжении есть только примитивы для линий и дуги окружности. Определим класс AShape следующим образом:

Таблица П-1. Характеристики языка Smalltalk

Абстракции	Переменные объектов Методы объектов Переменные классов Методы классов	Да Да Да Да
Ограничение доступа	Для переменных Для методов	Обособленные Общедоступные
Модульность	Виды модульности	Отсутствует
Иерархия	Наследование Обобщенные блоки Метаклассы	Простое Нет Да
Типирование	Строгое типирование Полиморфизм	Нет Да (простой)
Параллельность	Многозадачность	Да (определяется в классе)
Устойчивость	Устойчивость объектов	Нет

```
Object subclass: #AShape
  instanceVariableNames: 'theCenter'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Appendix'
```

```
initialize
```

```
«Initialize the shape»
```

```
theCenter <— Point new
```

```
setCenter: aPoint
```

```
«Set the center of the shape»
```

```
theCenter <— aPoint
```

```
center
```

```
«Return the center of the shape»
```

```
^theCenter
```

```
draw
```

```
«Draw the shape»
```

```
self subclassResponsibility
```

Теперь определим класс ACircle:

```
AShape subclass: #ACircle
  instanceVariableNames: 'theRadius'
  classVariableNames: ''
```

```

poolDictionaries: ''
category: 'Appendix'

setRadius: anInteger
    «Set the radius of the circle»

    theRadius <— anInteger

radius
    «Return the radius of the circle»

    ^theRadius

draw
    «Draw the circle»

    | anArc index |
    anArc <— Arc new.
    index <— 1.
    [index <= 4]
        whileTrue:
            [anArc
                center: theCenter
                radius: theRadius
                quadrant: index.
                anArc display.
                index <— index + 1]

```

Далее введем класс ARectangle:

```

AShape subclass: #ARectangle
instanceVariableNames: 'theWidth'
classVariableNames: ''
poolDictionaries: ''
category: 'Appendix'

setHeight: anInteger
    «Set the height of the rectangle»

    theHeight <— anInteger

setWidth: anInteger
    «Set the width of the rectangle»

    theWidth <— anInteger

height
    «Return the width of the rectangle»

    ^theHeight

width
    «Return the width of the rectangle»

    ^theWidth

```

```
draw
«Draw the rectangle»

! anLine upperLeftCorner !
anLine <— Line new.
upperLeftCorner <— theCenter x — (theWidth / 2) @ (theCenter y — (theHeight / 2)).
aLine beginPoint: upperLeftCorner.
aLine endPoint: upperLeftCorner x + theWidth @ upperCorner y.
aLine display.
aLine beginPoint: aLine endPoint.
aLine endPoint: upperLeftCorner x + theWidth @ (upperCorner y + theHeight).
aLine display.
aLine beginPoint: aLine endPoint.
aLine endPoint: upperLeftCorner x @ (upperCorner y + theHeight).
aLine display.
aLine beginPoint: aLine endPoint.
aLine endPoint: upperLeftCorner.
aLine display
```

Подкласс ASolidRectangle опишем следующим образом:

```
Arectangle subclass: #ASolidRectangle
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Appendix'
```

```
draw
«Draw the solid rectangle»

! upperLeftCorner lowerRightCorner !
super draw.
upperLeftCorner <— theCenter x - (theWidth quo: 2) + 1 @
                                   (theCenter y — (theHeight quo: 2) + 1).
lowerRightCorner <— upperLeftCorner x + theWidth — 1 @
                                   (upperLeftCorner y + theWidth — 1).

Display
  fill: (upperLeftCorner corner: lowerRightCorner)
  mask: Form gray
```

Литература

Основными руководствами по языку Smalltalk являются документы по Smalltalk-80: «The Language» Goldberg и Robson [6]; «The Interactive programming Environment» Goldberg [7]; «Bit of History», «Words of Advice» Krasner [8].

П.3. ЯЗЫК ОБЪЕКТ PASCAL

Основные положения

Язык Object Pascal создавался сотрудниками фирмы Apple Computer (некоторые из них были участниками проекта Smalltalk) совместно с Никла-

усом Виртом (Niklaus Wirth) — создателем языка Pascal. Непосредственным предшественником Object Pascal является Clascal (объектно-ориентированная версия Pascal для проекта Lisa). Object Pascal известен с 1986 г. и является первым объектно-ориентированным языком программирования, который нашел поддержку у программистов-пользователей Macintosh (MPW) и послужил основой для создания оболочки компьютеров семейства Apple. Для MPW создана библиотека классов, называемая MacApp и являющаяся основой для прикладных задач в соответствии с требованиями к интерфейсу пользователя Macintosh.

Таблица П-2. Характеристики языка Object Pascal

Абстракции	Переменные объектов Методы объектов Переменные классов Методы классов	Да Да Нет Нет
Ограничение доступа	Для переменных Для методов	Общедоступные Общедоступные
Модульность	Виды модульности	Модуль (интерфейс-реализация)
Иерархия	Наследование Обобщенные блоки Метаклассы	Простое Нет Нет
Типирование	Строгое типирование Полиморфизм	Да Да (простой)
Параллельность	Многозадачность	Нет
Устойчивость	Устойчивость объектов	Нет

На основе Object Pascal впоследствии созданы еще две версии объектно-ориентированного Pascal: Quick Pascal (фирмы Microsoft) и Turbo Pascal 5.x (фирмы Borland).

Обзор

Шмаккер утверждает, что «Object Pascal — «обглоданный» объектно-ориентированный язык. В нем отсутствуют методы класса, переменных класса, множественного наследования и метаклассов. Эти механизмы исключены специально, чтобы сделать язык простым для изучения программистами, начинающими изучать объектно-ориентированную методологию» [9]. В табл. П-2 приведены общие характеристики Object Pascal.

Пример

Принято размещать интерфейсную часть и реализацию классов в языке Object Pascal в разных файлах. Это связано с тем, что Object Pascal допу-

скаст отдельную компиляцию внешней и внутренней частей программного блока. Для задачи, связанной с выводом изображений, приведем интерфейсную часть классов:

```
unit UShapes; interface

uses
    UMacApp, UPrinting, IDialog, ToolUtils, Packages, UMacAppUtilities;

type
    TShape = object (TObject)
        fCenter: Point;
        procedure IShape;
        procedure SetCenter (TheCenter: Point);
        procedure Draw;
        function Center: Point;
    end;

    TCircle = object (TShape)
        fRadius: Integer;
        procedure IShape; override;
        procedure SetRadius (TheRadius: Integer);
        procedure Draw; override;
        function Radius: Integer;
    end;

    TRectangle = object (TShape)
        fHeight: Integer;
        fWidth: Integer;
        procedure IShape; override;
        procedure SetHeight (TheHeight: Integer);
        procedure SetWidth (TheWidth : Integer);
        procedure Draw; override;
        function Height: Integer;
        function Width: Integer;
    end;

    TSolidRectangle = object (TRectangle)
        procedure Draw; override;
    end;

implementation
    {$I UShapes.Inci.p}
end.
```

Отметим, что, несмотря на отсутствие формального ограничения доступа в Object Pascal, все поля предполагаются обособленными, а доступ к ним осуществляется через методы класса. В отдельном файле опишем реализацию классов с использованием графических средств библиотеки QuickDraw:

```

procedure TShape.IShape;
var
    APoint: Point;
begin
    SetPt (APoint, 0, 0);
    fCenter := APoint;
end;

procedure TShape.SetCenter (TheCenter : Point);
begin
    fCenter := TheCenter;
end;

procedure TShape.Draw;
begin
end;

function TShape.Center : Point;
begin
    Center := fCenter;
end;

procedure TCircle.IShape; override;
begin
    inherited IShape;
    fRadius := 0;
end;

procedure TCircle.SetRadius (TheRadius : Integer);
begin
    fRadius := TheRadius;
end;

procedure TCircle.Draw; override;
var
    ARect : Rect;
begin
    PenNormal;
    SetRect (ARect, fCenter.h, fCenter.v, fCenter.h, fCenter.v);
    InsetRect (ARect, -fRadius, -fRadius);
    FrameOval (ARect);
end;

function TCircle.Radius : Integer;
begin
    Radius := fRadius;
end;

procedure TRectangle.IShape; override;
begin
    inherited IShape;
    fHeight := 0;
    fWidth := 0;
end;

procedure TRectangle.SetHeight (TheHeight : Integer);

```

```

begin
    fHeight := TheHeight;
end;

procedure TRectangle.SetWidth (TheWidth : Integer);
begin
    fWidth := TheWidth;
end;

procedure TRectangle.Draw; override;
var
    ARect : Rect;
begin
    PenNormal;
    SetRect (ARect, fCenter.h, fCenter.v, fCenter.h, fCenter.v);
    InsetRect (ARect, -round(fWidth / 2), -round(fHeight / 2));
    FrameRect (ARect);
end;

function TRectangle.Height : Integer;
begin
    Height := fHeight;
end;

function TRectangle.Width : Integer;
begin
    Width := fWidth;
end;

procedure TSolidRectangle.Draw; override;
var
    ARect : Rect;
begin
    inherited Draw;
    SetRect (ARect, fCenter.h, fCenter.v, fCenter.h, fCenter.v);
    InsetRect (ARect, (1 - round(fWidth / 2)), (1 - round(fHeight / 2)));
    FillRect (ARect, Gray);
end;

```

Литература

Основным руководством по Object Pascal может служить Macintosh Programmer's Workshop Pascal 3.0 Reference [10].

П.4. Язык C++

Основные положения

Язык программирования C++ разработан сотрудником Bell Laboratories компании AT&T Берном Страустрапом (Bjarne Stroustrup). Непосредственным предшественником языка C++ является язык C with Classes, созданный тем же автором в 1980 г. Язык C with Classes в свою очередь был создан на основе языков C и Simula. C++ — это в значительной степени надстройка над C. В определенном смысле C++ можно назвать «улучшенным C», который обеспечивает контроль типов, перегрузку функций (переопределение) и ряд других свойств. Но главное состоит в том, что C++ ориентирован на объектное проектирование.

Известны две версии C++: 1.0 и 2.0. В версии 1.0 реализованы основные механизмы объектно-ориентированного программирования, такие, как простое наследование и полиморфизм, контроль типов и переопределение функций. В созданной в 1989 г. версии 2.0 нашли отражение многие дополнительные свойства (например, множественное наследование), полученные из опыта работы и обмена информацией с пользователями языка. В последующих версиях предполагается реализовать обобщенные модули (шаблоны) и обработку исключительных ситуаций.

Первые трансляторы C++ на основе препроцессора для языка C, названного *сfront*. Промежуточный код C мог быть использован в любых Unix системах. В настоящее время почти для всех типов ЭВМ созданы специальные компиляторы C++.

Обзор

Страустрап утверждает, что «C++ создавался с целью избавить автора и его друзей от необходимости программировать на ассемблере, C или других языках высокого уровня. Основной задачей было создание языка, на котором удобно писать хорошие программы и приятно работать программисту. Язык C++ никогда не проектировался на бумаге. Одновременно выполнялось его проектирование, документирование и реализация» [11]. C++ устранил многие недостатки C, добавив механизмы описания классов, контроль типов, переопределение функций, управление свободной памятью, постоянные типы, макроопределения (подставляемые функции), производные классы и виртуальные функции [12].

Характеристики C++ приведены в табл. П-3.

Пример

Вернемся снова к задаче вывода изображений. В языке C++ принято описывать интерфейсную часть классов в заголовочных файлах. Поэтому напомним:

```
struct Point {int X; int Y;};

class Shape {
public:
    Shape ();
    void SetCenter (Point ACenter);
    virtual void Draw () = 0;
    Point Center ();
private:
    Point TheCenter;
};

class Circle : public Shape {
public:
    Circle ();
    void SetRadius (int AnInteger);
    virtual void Draw ();
    int Radius ();
private:
    int TheRadius;
};

class Rectangle : public Shape {
public:
```

Таблица П-3. Характеристики языка C++

Абстракции	Переменные объектов Методы объектов Переменные классов Методы классов	Да Да Да Да
Ограничение доступа	Для переменных Для методов	Общедоступные, защищенные, обособленные Общедоступные защищенные, обособленные
Модульность	Виды модульности	Файл (заголовок-тело)
Иерархия	Наследование Обобщенные блоки Метаклассы	Множественная Нет Нет
Типирование	Строгое типирование Полиморфизм	Да Да (простой)
Параллельность	Многозадачность	Да (определяется в классе)
Устойчивость	Устойчивость объектов	Нет

```

Rectangle ();
void SetHeight (int AnInteger);
void SetWidth (int AnInteger);
virtual void Draw ();
int Height ();
int Width ();

private:
    int TheHeight;
    int TheWidth;
};

class SolidRectangle : public Rectangle {
public:
    virtual void Draw ();
};

```

Определения C++ не используют библиотеку классов. Предположим, что существуют программный интерфейс с X Windows и соответствующий ему объекты Display, Window, Graphics_Context. В отдельном файле напомним реализацию методов, перечисленных в заголовочном файле:

```

Shape::Shape () {
    TheCenter.X = 0;
    TheCenter.Y = 0;
};

```

```

void Shape::SetCenter (Point ACenter) {
    TheCenter = ACenter;
};

Point Shape::Center () {
    return TheCenter;
};

Circle::Circle () {
    TheRadius = 0;
};

void Circle::SetRadius (int AnInteger) {
    TheRadius = AnInteger;
};

void Circle::Draw () {
    int X = (Center () .X - TheRadius);
    int Y = (Center () .Y - TheRadius);
    XDrawArc (Display, Window, GraphicsContext, X, Y,
              (TheRadius * 2), (TheRadius * 2), 0, (360 * 64));
};

int Circle::Radius () {
    return TheRadius;
};

Rectangle::Rectangle () {
    TheHeight = 0;
    TheWidth = 0;
};

void Rectangle::SetHeight (int AnInteger) {
    TheHeight = AnInteger;
};

void Rectangle::SetWidth (int AnInteger) {
    TheWidth = AnInteger;
};

void Rectangle::Draw () {
    int X = (Center () .X - TheWidth / 2);
    int Y = (Center () .Y - TheHeight / 2);
    XDrawRectangle (Display, Window, GraphicsContext, X, Y, TheWidth, TheHeight);
};

int Rectangle::Height () {
    return TheHeight;
};

int Rectangle::Width () {
    return TheWidth;
};

void SolidRectangle::Draw () {
    Rectangle::Draw ();
    int X = (Center () .X - Width() / 2);
    int Y = (Center () .Y - Height() / 2);
    gc OldGraphicsContext = GraphicsContext;
    XSetForeground (Display, GraphicsContext, Grey);
    XDrawFilled (Display, Window, GraphicsContext, X, Y, Width(), Height());
    GraphicsContext = OldGraphicsContext;
};

```

Литература

Основным пособием по C++ является: UNIX System V AT&T C++ Language System, Release 2.0 Product Reference Manual [13]. Кроме того, Selected Readings [14], Release Notes [15] и Library Manual [16].

П.5. ЯЗЫК COMMON LISP OBJECT SYSTEM (CLOS)

Основные положения

В языке Lisp существует несколько десятков диалектов, включая MacLisp, Standard Lisp, SpiceLisp, S-1 Lisp, ZetaLisp, Nil, InterLisp и Scheme. В начале 80-х годов под воздействием идей объектно-ориентированного программирования возникла серия новых диалектов Lisp, многие из которых были ориентированы на представление знаний. Успех Ги Стила (Guy Steele) в стандартизации Common Lisp способствовал попытке стандартизировать объектно-ориентированные диалекты в 1986 г. Идея стандартизации была поддержана летней конференцией 1986 г. по ACM Lisp и функциональному программированию, в результате чего была создана специальная рабочая группа при комитете X3J13 ANSI (Комитет по стандартизации Common Lisp).

Таблица А-4. CLOS

Абстракции	Переменные объектов Методы объектов Переменные классов Методы классов	Да Да Да Да
Ограничение доступа	Для переменных Для методов	Чтение, запись, доступ Общедоступные
Модульность	Виды модульности	Пакет (монокитный)
Иерархия	Наследование Обобщенные блоки Метаклассы	Множественная Нет Да
Типирование	Строгое типирование Полиморфизм	Возможно Да (множественный)
Параллельность	Многозадачность	Да
Устойчивость	Устойчивость объектов	Косвенно

Поскольку новый диалект должен был стать надстройкой к Common Lisp, он получил название Common Lisp Object System (сокращенно CLOS). В комитет вошли Боб Матис и Ги Стил (Bob Mathis, Guy Steele). Возгла-

вил комитет Дэниел Бобров (Daniel Bobrow), а его членами стали Соня Кин, Линда де Мишель, Патрик Дассуд, Ричард Габриэль, Джеймс Кемпф, Грегор Кичазлес и Дэвид Мун (Sonya Keene, Linda de Michiel, Patrick Dussud, Richard Gabriel, James Kempf, Gregor Kiczales, David Moon).

Серьезное влияние на проект CLOS оказали языки NewFlavors и CommonLoops. После двухлетней работы в 1988 г. была опубликована полная спецификация CLOS.

Обзор

Кни (Keene) отмечает, что в проекте CLOS ставились три основные цели:

- * «Представление стандартного расширения языка, включающего все наиболее полезные приемы OOD.
- * Обеспечение эффективного и гибкого интерфейса программиста, позволяющего реализовать самые различные задачи.
- * Проект CLOS должен быть открытым для дальнейшего совершенствования и учета требований пользователя OOP» [17].

Обзор характеристик CLOS можно найти в табл. П-4. Не поддерживая прямо механизм устойчивости, CLOS имеет расширение с протоколом мета-объектов для реализации этого механизма [18].

Пример

В качестве примера рассмотрим снова задачу вывода изображений. Начнем с определения классов:

```
(defclass point ()
  ((x :initarg :x :accessor x :type integer)
   (y :initarg :y :accessor y :type integer)))

(defclass shape ()
  ((center :initarg :center :accessor center :type point)))

(defclass rectangle (shape)
  ((height :initarg :height :accessor height :type integer)
   (width :initarg :width :accessor width :type integer)))

(defclass solid-rectangle (rectangle)
  ())

(defclass circle (shape)
  ((radius :initarg :radius :accessor radius :type integer)))
```

Приведем методы для прямоугольника:

```
(defmethod left ((r rectangle))
  (- (x (center r)) (/ (width r) 2)))

(defmethod right ((r rectangle))
  (+ (x (center r)) (/ (width r) 2)))

(defmethod top ((r rectangle))
  (+ (y (center r)) (/ (height r) 2)))

(defmethod bottom ((r rectangle))
  (- (y (center r)) (/ (height r) 2)))
```

Данное описание не использует библиотеку классов. Допустим существование программного протокола с ресурсами, аналогичными Object Pascal QuickDraw, включая рисование линий, дуг и штриховку областей. Введем также класс windows, обозначающий изображение на дисплее. Теперь напомним следующие методы:

```
(defgeneric draw (shape surface))

(defmethod draw ((s shape) window) ())

(defmethod draw ((r rectangle) window)
  (draw-line window (left r) (bottom r) (left r) (top r))
  (draw-line window (left r) (bottom r) (right r) (bottom r))
  (draw-line window (right r) (bottom r) (right r) (top r))
  (draw-line window (left r) (top r) (left r) (top r)))

(defmethod draw :after ((sr solid-rectangle) window)
  (fill-area window *gray-shade* (left sr) (bottom sr) (right sr) (top sr)))

(defmethod draw ((c circle) window)
  (draw-arc gw (center c) (radius c) 0 360))
```

Литература

Основным руководством по CLOS может служить Common Lisp Object Specification [19].

П.6. ЯЗЫК ADA

Основные положения

Министерство обороны США, вероятно, имеет самое большое число компьютеров во всем мире. В середине 70-х годов его программисты оказались в критической ситуации: проекты не укладывались в установленные сроки и бюджет, а заданных характеристик не удавалось реализовать. Стало очевидно, что ситуация может только ухудшиться, поэтому министерство обороны финансировало проект создания единого языка высокого уровня. В этом смысле язык Ada является одним из первых языков программирования промышленного уровня. Исходные требования были сформулированы в 1975 г. в документе Стильмана (Steelman) и реализованы в 1978 г. Был объявлен международный конкурс, на который откликнулось 17 участников, и был выбран один проект, в реализации которого участвовали сотни ученых всех стран мира.

Проект-победитель вначале носил условное наименование Green (анонимное кодовое название), а позднее получил имя Ada в честь Ada Augusta, графини Lovelace, которая известна по ее предсказаниям относительно будущего компьютеризации. Основным разработчиком языка был Джоан Ичблан (Jean Ichbian) из Франции. В команду разработчиков входили: Bernd Krieg-Brueckner, Brian Wichmann, Henry Ledgard, Jean-Claude Heliard, Jean-Loup Gailly, Jean-Raymond Abrial, John Barnes, Mike Woodger, Olivier Roubine, S.A. Schuman и S.C. Vestal.

Непосредственными предшественниками Ada являются Pascal и его производные, включая Euclid, Lis, Mesa, Modula и Sue. Были использованы некоторые идеи ALGOL-68, Simula, CLU и Alphard. Стандарт ANSI для Ada был окончательно сформирован в 1983 г. Трансляторы Ada реализованы для

всех основных архитектур ЭВМ, использующих наборы инструкций. Язык Ada в настоящее время используется во многих государственных и коммерческих проектах (например, проект совместной европейско-американской космической станции). Стандарты ANSI пересматриваются каждые пять лет, поэтому в настоящее время действует проект Ada 9х. Исходные определения языка со временем изменились для обеспечения большей ясности, для устранения дефектов и исправления ошибок. В настоящем виде Ada является объектным, но не объектно-ориентированным языком. Однако есть множество предложений по расширению языка Ada до уровня объектно-ориентированного.

Таблица II-5. Характеристики языка Ada

Абстракции	Переменные объектов Методы объектов Переменные классов Методы классов	Да Да Нет Нет
Ограничение доступа	Для переменных Для методов	Общедоступные, обособленные Общедоступные, обособленные
Модульность	Виды модульности	Пакет (спецификация-тело)
Иерархия	Наследование Обобщенные блоки Метаклассы	Нет Да Нет
Типирование	Строгое типирование Полиморфизм	Да Нет
Параллельность	Многозадачность	Да (определяется в языке)
Устойчивость	Устойчивость объектов	Нет

Обзор

Перед разработчиками Ada стояли три важные цели:

- * Надежность и эксплуатационные качества программ.
- * Программирование — это деятельность человека.
- * Эффективность [20].

В табл. II-5 приведены основные характеристики языка Ada с точки зрения объектного подхода.

Пример

Рассмотрим снова задачи вывода изображений. Для языка Ada принято описывать классы в спецификации пакета. В нашем примере каждый класс размещается в отдельном пакете:

```

package Points is
    type Point is record
        X : Natural;
        Y : Natural;
    end record;
end Points;

with Points;
package Circle is
    type Circle is private;

    procedure Set_Center      (The_Center : in out Circle; The_Center : in Point);
    procedure Set_Radius      (The_Center : in out Circle; The_Radius : in Natural);
    procedure Draw            (The_Center : in Circle);

    function Center_Of (The_Circle : in Circle) return Point;
    function Radius_Of (The_Circle : in Circle) return Natural;

private
    type Circle is record
        Center : Point;
        Radius : Natural;
    end record;
end Circle;

with Points;
package Rectangle is
    type Rectangle (Is_Solid : Boolean := False) is private;

    procedure Set_Center      (The_Rectangle : in out Rectangle;
                               The_Center    : in Point);

    procedure Set_Height      (The_Rectangle : in out Rectangle;
                               The_Height    : in Natural);

    procedure Set_Width       (The_Rectangle : in out Rectangle;
                               The_Width     : in out Natural);
    procedure Draw            (The_Rectangle : in Rectangle);

    function Center_Of (The_Rectangle : in Rectangle) return Point;
    function Height_Of (The_Rectangle : in Rectangle) return Natural;
    function Width_Of  (The_Rectangle : in Rectangle) return Natural;

private
    type Rectangle (Is_Solid : Boolean) is record
        Center : Point;
        Height : Natural;
        Width  : Natural;
    end record;
end Rectangles;

```

Реализация пакетов выполнена на основе связи языка Ada с XWindows с помощью глобальных переменных Display, Window, Graphics_Context:

```

procedure Set_Center (The_Circle : in out Circle; The_Center : in Point) is
begin
    The_Circle.Center := The_Center;
end Set_Center;

```

```

procedure Set_Radius (The_Circle : in out Circle; The_Radius : in Natural) is
begin
    The_Circle.Radius := The_Radius;
end Set_Radius;

procedure Draw (The_Circle : in Circle) is
    X : Integer := The_Circle.Center.X - The_Circle.The_Radius;
    Y : Integer := The_Circle.Center.Y - The_Circle.The_Radius;
begin
    XDrawArc      (Display, Window, Graphics_Context, X,Y,
                  (The_Circle.The_Radius * 2), (The_Circle.The_Radius * 2),
                  0, (360 * 64));
end Draw;

function Center_Of (The_Circle : in Circle) return Point is
begin
    return The_Circle.Center;
end Center_Of;

function Radius_Of (The_Circle : in Circle) return Natural is
begin
    return The_Circle.Radius;
end Radius_Of;

procedure Set_Center      (The_Rectangle : in out Rectangle;
                          The_Center    : in Point) is
begin
    The_Rectangle.Center := The_Center;
end Set_Center;

procedure Set_Height      (The_Rectangle : in out Rectangle;
                          The_Height    : in Natural) is
begin
    The_Rectangle.Height := The_Height;
end Set_Height;

procedure Set_Width       (The_Rectangle : in out Rectangle;
                          The_Width     : in out Natural) is
begin
    The_Rectangle.Width := The_Width;
end Set_Width;

procedure Draw (The_Rectangle : in Rectangle) is
    X : Integer := The_Rectangle.The_Center.X - (The_Rectangle.The_Width;
    Y : Integer := The_Rectangle.The_Center.Y - (The_Rectangle.The_Height;
    Old_Graphics_Context : GC := Graphics_Context;
begin
    XDrawRectangle (Display, Window, Graphics_Context, X,Y,
                  (The_Rectangle.The_Width, (The_Rectangle.The_Height);
    if The_Rectangle.Is_Solid then
        XSetForeground (Display, Graphics_Context, Grey);
        XDrawFilled    (Display, Window, Graphics_Context, X,Y,
                      The_Rectangle.The_Width, The_Rectangle.The_Height);
        Graphics_Context := Old_Graphics_Context;
    end if;
end Draw;

function Center_Of (The_Rectangle : in Rectangle) return Point is
begin
    return The_Rectangle.Center;
end Center_Of;
function Height_Of (The_Rectangle : in Rectangle) return Natural is
begin

```

```

    return The_Rectangle.Height;
end Height_Of;

function Width_Of (The_Rectangle : in Rectangle) return Natural is
begin
    return The_Rectangle.Width;
end Width_Of;

```

Литература

Основным руководством по языку Ada является Reference Manual for the Ada Programming Language [21].

П.7. ДРУГИЕ ЯЗЫКИ ООР

В работе Саундерса (Saunders) [22] дан обзор более чем восьмидесяти языков ООР. Автор группирует эти языки по семи характеристикам [23]:

- | | |
|---------------------------|---|
| * Воздействие | Языки, поддерживающие механизм делегирования. |
| * Параллельность | Объектно-ориентированные языки, реализующие параллельность. |
| * Распределенность | Наличие распределенных объектов. |
| * Фреймовые структуры | Реализация теории фреймов. |
| * Смешение | Расширения традиционных языков до уровня ООР. |
| * Использование Smalltalk | Диалекты Smalltalk. |
| * Ориентация | Область применения ООР. |
| * Прочее | Языки ООР, не относящиеся ни к одному из категорий. |

На рис. П-2 перечислены основные языки ООР, а в библиографии приведены соответствующие источники информации.

ABCL/1	Concurrent Smalltalk	Lore	Plasma II
ABE	CSSA	Mace	POOL-T
Acorn	CST	MELD	PROCOL
Act/1	Director	Mjolner	Quick Pascal
Act/2	Distributer Smalltalk	ModPascal	Quicktalk
Act/3	Eiffel	Neon	ROSS
Actor	Emerald	New Flavora	SAST
Actora	ExperCommonLisp	NIL	SCOOP
Actra	Extended Smalltalk	O-CPU	SCOOPS
Ada	Felix Pascal	OakLisp	Self
Argue	Flavora	Oberon	Simula
ART	FOOPlog	Object Assembler	SINA
Berkeley Smalltalk	FOOPS	Object Cobol	Smalltalk
Beta	FRL	Object Lisp	Smalltalk AT
Blaze	Galileo	Object Logo	Smalltalk V
Brouhaha	Garp	Object Oberon	Smallworld
C with Classes	GLISP	Object Pascal	SPOOL
C ++	Gypay	Objective-C	SR
C.talk	Hybrid	ObjVLisp	SRL
Cantor	Inheritance	OOPC	STROBE
Classic	InnovAda	OOPC+	T
Classic Ada	Intermission	OPAL	Trellis/Owl
CLOS	Jasmine	Orbit	Turbo Pascal 5.x
Cluster 86	KL-One	Orient84/K	Uniform
Common Loops	KRL	OTM	UNITS
Common Objects	KRS	PCOL	Vulcan
Common ORBIT	Little Smalltalk	PIE	XLISP
Concurrent Prolog	LOOPs	PLLL	Zoom/VM

Рис. П-2. Объектные и объектно-ориентированные языки программирования.

Литература

Введение

Mills, H. 1985. *DPMA and Human Productivity*. Houston, TX: Data Processing Management Association.

Часть I. КОНЦЕПЦИИ

Wagner, J. 1986. *The Search for Signs of Intelligent Life in the Universe*. New York, NY: Harper and Row, p. 202. By permission of ICM, Inc.

Глава I. Сложность

- [1] Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, vol. 20 (4), p. 12.
- [2] Peters, L. 1981. *Software Design*. New York, NY: Yourdon Press, p. 22.
- [3] Brooks, F. April 1987. No Silver Bullet, p. 11.
- [4] Parnas, D. July 1985. Why Software Is Unreliable. *Software Aspects of Strategic Defense Systems*. Victoria, Canada: University of Victoria, Report DCS-47-IR.
- [5] Peter, L. 1986. *The Peter Pyramid*. New York, NY: William Morrow, p. 153.
- [6] Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM*, vol. 28 (6), p. 596.
- [7] Simon, H. 1982. *The Sciences of the Artificial*. Cambridge, MA: The MIT Press, p.218.
- [8] Ibid., p. 217.
- [9] Ibid., p. 221.
- [10] Ibid., p. 209.
- [11] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. 2nd ed. Ann Arbor, MI: The General Systemantics Press, p. 65.
- [12] Miller, G. March 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, vol. 63 (2), p. 86.
- [13] Simon. *Sciences*, p. 81.
- [14] Dijkstra, E. 1979. Programming Considered as a Human Activity. *Classics in Software Engineering*. New York, NY: Yourdon Press, p.5.
- [15] Parnas, D. December 1985. Software Aspects of Strategic Defense Systems. *Communications of the ACM*, vol. 28 (12), p. 1328.
- [16] Tsai, J., and Ridge, J. November 1988. Intelligent Support for Specifications Transformation. *IEEE Software*, vol. 5 (6), p. 34.
- [17] Langdon, G. 1982. *Computer Design*. San Jose, CA: Computeach Press, p. 6.
- [18] Miller. Magical Number, p.95.
- [19] Shaw, M. 1981. *ALPHARD: Form and Content*. New York, NY: Springer-Verlag, p. 6.
- [20] Stein, J. March 1988. Object-Oriented Programming and Database Design. *Dr. Dobbs's Journal of Software Tools for the Professional Programmer*, No. 137, p.18.

- [21] Peters. *Software Design*.
- [22] Yau, S. and Tsai, J. June 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering*, vol. SE-12 (6).
- [23] Teledyne Brown Engineering. *Software Methodology Catalog*, Report MC87-COMM/ADP-0036. October 1987. Tinton Falls, NJ.
- [24] Sommerville, I. 1985. *Software Engineering*. 2nd ed. Workingham, England: Addison-Wesley, p. 68.
- [25] Yourdon, E., and Constantine, L. 1979. *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall.
- [26] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold.
- [27] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.
- [28] Wirth, N. January 1983. Program Development by Stepwise Refinement. *Communications of the ACM*, vol. 26 (1).
- [29] Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall.
- [30] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press.
- [31] Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis*. Orlando, FL: Academic Press.
- [32] Jackson, M. 1975. *Principles of Program Design*. Orlando, FL: Academic Press.
- [33] Jackson, M. 1983. *System Development*. Englewood Cliffs, NJ: Prentice-Hall.
- [34] Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.
- [35] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, p. 80.
- [36] Petroski, H. 1985. *To Engineer Is Human*. St Martin's Press: New York, p. 40.
- [37] Mostow, J. Spring 1985. Toward Better Models of the Design Process. *AI Magazine*, vol. 6 (1), p. 44.
- [38] Eastman, N. 1984. Software Engineering and Technology. *Technical Directions*, vol. 10 (1): Bethesda, MD: IBM Federal Systems Division, p. 5.
- [39] Brooks. No Silver Bullet, p. 10.

Глава 2. Объектный подход

- [1] Rentsch, T. September 1982. Object-Oriented Programming. *SIGPLAN Notices*, vol. 17 (12), p. 51.
- [2] Wegner, P. June 1981. *The Ada Programming Language and Environment*. Unpublished draft.
- [3] Abbott, R. August 1987. Knowledge Abstraction. *Communications of the ACM*, vol. 30 (8), p. 664.
- [4] Ibid., p. 664.
- [5] Shankar, K. 1984. Data Design: Types, Structures, and Abstractions. *Handbook of Software Engineering*. New York, NY: Van Nostrand Reinhold, p. 253.

- [6] *Macintosh MacApp 1.1.1 Programmer's Reference*. 1986. Cupertino, CA: Apple Computer, p. 2.
- [7] Bhaskar, K. October 1983. How Object-Oriented Is Your System? *SIGPLAN Notices*, vol. 18 (10), p. 8.
- [8] Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations, *AI Magazine*, vol. 6 (4), p. 41.
- [9] Yonezawa, A., and Tokoro, M. 1987. Object-Oriented Concurrent Programming: An Introduction, in *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press, p. 2.
- [10] Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press, p. 13.
- [11] Ramamoorthy, C., and Sheu, P. Fall 1988. Object-Oriented Systems. *IEEE Expert*, vol. 3 (3), p. 14.
- [12] Myers, G. 1982. *Advances in Computer Architecture*. 2nd ed. New York, NY: John Wiley and Sons, p. 58.
- [13] Levy. *Capability-Based Computer*.
- [14] Kavi, K., and Chen, D. 1987. Architectural Support for Object-Oriented Languages. *Proceedings of the Thirty-second IEEE Computer Society International Conference* IEEE.
- [15] *APX 432 Object Primer*. 1981. Santa Clara, CA: Intel Corporation.
- [16] Dally, W. J., and Kajiya, J. T. March 1985. An Object-Oriented Architecture. *SIGARCH Newsletter*, vol. 13 (3).
- [17] Dahlby, S., Henry, G., Reynolds, D., and Taylor, P. 1982. The IBM System/38: A High Level Machine, in *Computer Structures: Principles and Examples*. New York, NY: McGraw-Hill.
- [18] Dijkstra, E. May 1968. The Structure of the "THE" Multiprogramming System. *Communications of the ACM*, vol. 11 (5).
- [19] Pashtan, A. 1982. Object-Oriented Operating Systems: An Emerging Design Methodology. *Proceedings of the ACM '82 Conference*. ACM.
- [20] Parnas, D. 1979. On the Criteria to Be Used in Decomposing Systems into Modules, in *Classics in Software Engineering*. New York, NY: Yourdon Press.
- [21] Liskov, B., and Zilles, S. 1977. An Introduction to Formal Specifications of Data Abstractions. *Current Trends in Programming Methodology: Software Specification and Design*, vol. 1. Englewood Cliffs, NJ: Prentice-Hall.
- [22] Guttag, J. 1980. Abstract Data Types and the Development of Data Structures, in *Programming Language Design*. New York, NY: Computer Society Press.
- [23] Shaw. Abstraction Techniques.
- [24] Nygaard, K., and Dahl, O-J. 1981. The Development of the Simula Languages, in *History of Programming Languages*. New York, NY: Academic Press, p. 460.
- [25] Atkinson, M., and Buneman, P. June 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, vol. 19 (2), p. 105.
- [26] Rumbaugh, J. April 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM*, vol. 31 (4), p. 415.

- [27] Chen, P. March 1976. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, vol. 1(1).
- [28] Barr, A., and Feigenbaum, E. 1981. *The Handbook of Artificial Intelligence*. Vol. 1. Los Altos, CA: William Kaufmann, p. 216.
- [29] Stillings, N., Feinstein, M., Garfield, J., Rissland, E., Rosenbaum, D., Weisler, S., Baker-Ward, L. 1987. *Cognitive Science: An Introduction*. Cambridge, MA: The MIT Press, p. 305.
- [30] Rand, Ayn. 1979. *Introduction to Objectivist Epistemology*. New York, NY: New American Library.
- [31] Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster.
- [32] Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software. *Operating Systems*. New York, NY: Springer-Verlag, p. 8.
- [33] Stroustrup, B. May 1988. What Is Object-Oriented Programming? *IEEE Software*, vol. 5 (3), p. 10.
- [34] Cardelli, L., and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. *ACM Computing Surveys*, vol. 17 (4), p. 481.
- [35] DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.
- [36] Yourdon, E. 1989. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [37] Gane, C., and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [38] Ward, P., and Mellor, S. 1985. *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Yourdon Press.
- [39] Hatley, D., and Pirbhaj, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.
- [40] Jenkins, M., and Glasgow, J. January 1986. Programming Styles in Nial. *IEEE Software*, vol. 3 (1), p. 48.
- [41] Bobrow, D., and Stefik, M. February 1986. Perspectives on Artificial Intelligence Programming. *Science*, vol. 231, p. 951.
- [42] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press, p. 83.
- [43] Shaw, M. October 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software*, vol. 1 (4), p. 10.
- [44] Berzins, V., Gray, M., and Naumann, D. May 1986. Abstraction-Based Software Development. *Communications of the ACM*, vol. 29 (5), p. 403.
- [45] Abelson, H., and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press, p. 126.
- [46] Ibid., p. 132.
- [47] Seidewitz, E., and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. NASA Lyndon B. Johnson Space Center, TX: NASA, p. D.4.6.4.
- [48] Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, p. 9.

- [49] Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. *IEEE Transactions on Software Engineering*, vol. SE-13 (7), p. 820.
- [50] Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, p. 180.
- [51] Liskov, B. May 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices*, vol. 23 (5), p. 19.
- [52] Britton, K., and Parnas, D. December 8, 1981. *A-7E Software Module Guide*. Washington, D.C. Naval Research Laboratory, Report 4702, p. 24.
- [53] Stroustrup, B. 1988. Private communication.
- [54] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold, p. 21.
- [55] Liskov, B. 1980. A Design Methodology for Reliable Software Systems, in *Tutorial on Software Design Techniques*. 3rd ed. New York, NY: IEEE Computer Society, p. 66.
- [56] Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys*, vol. 10 (2), p. 20.
- [57] Parnas, D., Clements, P., and Weiss, D. March 1985. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, vol. SE-11 (3), p. 260.
- [58] Britton and Parnas. *A-7E Software*, p. 2.
- [59] Parnas, D., Clements, P., and Weiss, D. 1983. Enhancing Reusability with Information Hiding. *Proceedings of the Workshop on Reusability in Programming*, Stratford, CT: ITT Programming, p. 241.
- [60] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall, p. 47.
- [61] Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, p. 69.
- [62] Danforth, S., and Tomlinson, C. March 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, vol. 20 (1), p. 34.
- [63] Liskov. 1988, p. 23.
- [64] As quoted in Liskov. 1980, p. 67.
- [65] Zilles, S. 1984. Types, Algebras, and Modelling, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York, NY: Springer-Verlag, p. 442.
- [66] Borning, A., and Ingalls, D. 1982. A Type Declaration and Inference System for Smalltalk. Palo Alto, CA: Xerox Palo Alto Research Center, p. 134.
- [67] Wegner, P. October 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices*, vol. 22 (12), p. 171.
- [68] Tesler, L. August 1981. The Smalltalk Environment. *Byte*, vol. 6 (8), p. 142.
- [69] Borning and Ingalls. Type Declaration, p. 133.
- [70] Thomas, D. March 1989. What's in an Object? *Byte*, vol. 14 (3), p. 232.
- [71] Lim, J., and Johnson, R. April 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices*, vol. 24 (4), p. 165.
- [72] Ibid., p. 165.

- [73] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. July 1986. *Distribution and Abstract Types in Emerald*. Report 86-02-04. Seattle, WA: University of Washington, p. 3.
- [74] Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. April 1989. *SIGPLAN Notices*, vol. 24 (4), p. 1.
- [75] Atkinson, M., Bailey, P., Chisholm, K., Cockshott, P., and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal*, vol. 26 (4), p. 360.
- [76] Khoshafian, S., and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices*, vol. 21 (11), p. 409.
- [77] *Vbase Technical Overview*. September 1987. Billerica, MA: Ontologic, p. 4.
- [78] Stroustrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM, p. 14.
- [79] Meyer. *Object-Oriented Software*, p. 233.
- [80] Robson, D. August 1981. Object-Oriented Software Systems. *Byte*, vol. 6 (8), p. 74.

Глава 3. Классы и объекты

- [1] Lefrancois, G. 1977. *Of Children: An Introduction to Child Development*. 2nd ed. Belmont, CA: Wadsworth, p. 244-246.
- [2] Nygaard, K., and Dahl, O-J. 1981. The Development of the Simula Languages, in *History of Programming Languages*. New York, NY: Academic Press, p. 462.
- [3] Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, vol. 4 (5), p. 73.
- [4] Smith, M., and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects*. Seattle, WA: Boeing Commercial Airplane Support Division, p. 132.
- [5] Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, p. 29.
- [6] MacLennan, B. December 1982. Values and Objects in Programming Languages. *SIGPLAN Notices*, vol. 17 (12), p. 78.
- [7] Brodie, M., and Ridjanovic, D. 1984. On the Design and Specification of Database Transactions, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York, NY: Springer-Verlag, p. 288.
- [8] Lippman, S. 1989. *C++ Primer*. Reading, MA: Addison-Wesley, p. 185.
- [9] *Macintosh MacApp 1.1.1 Programmer's Reference*. 1986. Cupertino, CA: Apple Computer, p. 4.
- [10] Khoshafian, S., and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices*, vol. 21 (11), p. 406.
- [11] *Ibid.*, p. 406.
- [12] Ingalls, D. 1981. Design Principles behind Smalltalk. *Byte*, vol. 6 (8), p. 290.
- [13] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. 2nd ed. Ann Arbor, MI: The General Systemantics Press, p. 158.
- [14] Seidewitz, E., and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on*

Ada Programming Language Applications for the NASA Space Station. NASA Lyndon B. Johnson Space Center, TX: NASA, p. D.4.6.4.

- [15] Webster's Third New International Dictionary of the English Language, unabridged. 1986. Chicago, Illinois: Merriam-Webster.
- [16] Meyer, B. 1987. *Programming as Contracting*. Report TR-EI-12/CO. Goleta, CA: Interactive Software Engineering.
- [17] Snyder, A. November 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. *SIGPLAN Notices*, vol. 21 (11).
- [18] LaLonde, W. April 1989. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems*, vol. 11 (2), p. 214.
- [19] Rumbaugh, J. April 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM*, vol. 31 (4), p. 417.
- [20] Lieberman, H. November 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *SIGPLAN Notices*, vol. 21 (11).
- [21] Brachman, R. October 1983. What IS-A Is and Isn't: An Analysis of Taxonomic Link in Semantic Networks. *IEEE Computer*, vol. 16 (10), p. 30.
- [22] Micallef, J. April/May 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, vol. (1), p. 15.
- [23] Snyder. Encapsulation, p. 39.
- [24] Cardelli, L., and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. *ACM Computing Surveys*, vol. 17 (4), p. 475.
- [25] As quoted in Harland, D., Szyplewski, M., and Wainwright, J. October 1985. An Alternative View of Polymorphism. *SIGPLAN Notices*, vol. 20 (10).
- [26] Kaplan, S., and Johnson, R. July 21, 1986. *Designing and Implementing for Reuse*. Urbana, IL: University of Illinois, Department of Computer Science, p. 8.
- [27] Deutsch, P. 1983. Efficient Implementation of the Smalltalk-80 System, in *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, p. 300.
- [28] Ibid., p. 299.
- [29] Duff, C. August 1986. Designing an Efficient Language. *Byte*, vol. 11 (8), p. 216.
- [30] Stroustrup, B. 1988. Private communication.
- [31] Stroustrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, New Mexico, p. 8.
- [32] Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley, p. 44.
- [33] Winston, P., and Horn, B. 1989. *Lisp*. 3rd ed. Reading, MA: Addison-Wesley, p. 510.
- [34] Keene, p. 11.
- [35] Micallef, J. April/May 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, vol. 1 (1), p. 25.
- [36] Snyder. Encapsulation, p. 41.
- [37] Vlissides, J., and Linton, M. 1988. Applying Object-Oriented Design to Structured Graphics. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association, p. 93.

- [38] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall, p. 274.
- [39] Keene. *Object-Oriented Programming*, p. 118.
- [40] Snyder. Encapsulation, p. 43.
- [41] Hendler, J. October 1986. Enhancement for Multiple Inheritance. *SIGPLAN Notices*, vol. 21 (10), p. 100.
- [42] Booch, G. 1987. *Software Components with Ada*. Menlo Park, CA: Benjamin/Cummings.
- [43] Stroustrup, 1987, p. 3.
- [44] Stroustrup, B. 1988. Parameterized Types for C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association, p. 1.
- [45] Ibid.
- [46] Meyer, B. November 1986. Genericity versus Inheritance. *SIGPLAN Notices*, vol. 21 (11), p. 402.
- [47] Stroustrup. 1988, p. 4.
- [48] Robson, D. August 1981. Object-Oriented Software Systems. *Byte*, vol. 6 (8), p. 86.
- [49] Goldberg, A., and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley, p. 287.
- [50] Ingalls, D. August 1981. Design Principles behind Smalltalk. *Byte*, vol. 6 (8), p. 286.
- [51] Stevens, W., Myers, G., and Constantine, L. 1979. Structured Design, in *Classics in Software Engineering*. New York, NY: Yourdon Press, p. 209.
- [52] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press, p. 59.
- [53] Meyer. 1987, p. 4.
- [54] Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, vol. 4 (5), p. 74.
- [55] Sakkinen, M. December 1988. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices*, vol. 23 (12), p. 38.
- [56] Lea, D. August 12, 1988. *User's Guide to GNU C++ Library*. Cambridge, MA: Free Software Foundation, p. 12.
- [57] Ibid.
- [58] Meyer. 1988, p. 332.
- [59] Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall, p. 37.
- [60] Keene. *Object-Oriented Programming*, p. 68.
- [61] Parnas, D., Clements, P., and Weiss, D. 1989. Enhancing Reusability with Information Hiding. *Software Reusability*. New York, NY: ACM Press, p. 143.

Глава 4. Классификация

- [1] As quoted in Swaine, M. June 1988. Programming Paradigms. *Dr. Dobbs Journal of Software Tools*, No. 140, p. 110.
- [2] Michalski, R. and Stepp, R. 1983. Learning from Observation: Conceptual Clustering, in *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga, p. 332.

- [13] Darwin, C. 1984. *The Origin of Species. Vol. 49 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 207.
- [14] *The New Encyclopedia Britannica*. 1985. Chicago, IL: Encyclopedia Britannica. vol. 3, p. 356.
- [15] May, R. September 16, 1988. How Many Species Are There on Earth? *Science*, vol. 241, p. 1441.
- [16] As quoted in Lewin, R. November 4, 1988. Family Relationships Are a Biological Conundrum. *Science*, vol. 242, p. 671.
- [17] *The New Encyclopedia Britannica*, vol. 3, p. 156.
- [18] Descartes, R. 1984. *Rules for the Direction of the Mind. Vol. 31 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 32.
- [19] Shaw, M. May 1989. Larger Scale Systems Require Higher-Level Abstractions. *SIGSOFT Engineering Notes*, vol. 14 (3), p. 143.
- [100] Goldstein, T. May 1989. The Object-Oriented Programmer. *The C++ Report*, vol. 1 (5).
- [111] Coombs, C., Raiffa, H., and Thrall, R. 1954. Some Views on Mathematical Models and Measurement Theory. *Psychological Review*, vol. 61 (2), p. 132.
- [112] Flood, R., and Carson, E. 1988. *Dealing with Complexity*. New York, NY: Plenum Press, p. 8.
- [113] Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur, p. 23.
- [114] Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group, p. 11.
- [115] Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley, p. 16.
- [116] Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. Chicago, IL: The University of Chicago Press, p. 161.
- [117] Wegner, P. 1987. The Object-Oriented Classification Paradigm, in *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press, p. 480.
- [118] Aquinas, T. 1984. *Summa Theologica. Vol. 19 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 71.
- [119] Stepp, R., and Michalski, R. February 1986. Conceptual Clustering of Structured Objects: A Goal-Oriented Approach. *Artificial Intelligence*, vol. 28 (1), p. 53.
- [120] Maier, H. 1969. *Three Theories of Child Development: The Contributions of Erik H. Erickson, Jean Piaget, and Robert R. Sears, and Their Applications*. New York, NY: Harper and Row, p. 111.
- [121] Lakoff. *Women, Fire*, p. 32.
- [122] Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster, p. 199.
- [123] *The Great Ideas: A Syntopicon of Great Books of the Western World*. 1984. Vol. 1 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 293.
- [124] Stepp, p. 44.
- [125] Lakoff. *Women, Fire*, p. 7.
- [126] *Ibid.*, p. 16.

- [27] Meyer, private communication.
- [28] Shlaer, S., and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, p. 15.
- [29] Ross, R. 1987. *Entity Modeling: Techniques and Application*. Boston, MA: Database Research Group, p. 9.
- [30] Coad, P., and Yourdon, E. 1990. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall, p. 62.
- [31] Arango, G. May 1989. Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes*, vol. 14 (3), p. 153.
- [32] Moore, J., and Bailin, S. 1988. *Position Paper on Domain Analysis*. Laurel, MD: CTA Incorporated, p. 2.
- [33] Abbott, R. November 1983. Program Design by Informal English Descriptions. *Communications of the ACM*, vol. 26 (11).
- [34] Saeki, M., Horai, H., and Enomoto, H. May 1989. Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- [35] McNamee, S., and Palmer, J. 1984. *Essential Systems Analysis*. New York, NY: Yourdon Press, p. 267.
- [36] Ward, P., and Mellor, S. 1985. *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Yourdon Press.
- [37] Seidewitz, E., and Stark, M. August 1986. *General Object-Oriented Software Development*, Report SEL-86-002. Greenbelt, MD: NASA Goddard Space Flight Center, p. 5-2.
- [38] Seidewitz, E. 1990. Private communication.
- [39] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, p. 77.
- [40] Thomas, D. May/June 1989. In Search of an Object-Oriented Development Process. *Journal of Object-Oriented Programming*, vol. 2 (1), p. 61.
- [41] Stroustrup, B. 1986. *The C++ Programming Language*. Reading, MA: Addison-Wesley, p. 7.
- [42] Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, vol. 4 (5), p. 75.
- [43] Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations, *AI Magazine*, vol. 6 (4), p. 60.
- [44] Stefik and Bobrow. Object-Oriented Programming, p. 58.
- [45] Jackson, M. 1983. *System Development*. Englewood Cliffs, NJ: Prentice-Hall.
- [46] Ibid., p. 4.
- [47] Adams, S. July 1986. MetaMethods: The MVC Paradigm, in *HOOPLA: Hooray for Object-Oriented Programming Languages*. Everett, WA: Object-Oriented Programming for Smalltalk Applications Developers Association, vol. 1 (4), p. 6.
- [48] Russo, V., Johnston, G., and Campbell, R. September 1988. Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices*, vol. 23 (11), p. 249.

- [49] Englemore, R., and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley, p. v.

Часть II. МЕТОДОЛОГИЯ

Petroski, H. 1985. *To Engineer is Human*. New York, NY: St Martin's Press, p. 73.

Глава 5. Система обозначений

- [1] Shear, D. December 8, 1988. CASE Shows Promise, but Confusion Still Exists. *EDN*, vol. 33 (25), p. 168.
- [2] Whitehead, A. 1958. *An Introduction to Mathematics*. New York, NY: Oxford University Press.
- [3] Defense Science Board. *Report of the Defense Science Board Task Force on Military Software*. September 1987. Washington, D.C.: Office of the Undersecretary of Defense for Acquisition, p. 8.
- [4] Kleyn, M., and Gingrich, P. September 1988. GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views. *SIGPLAN Notices*, vol. 23 (11), p. 192.
- [5] Weinberg, G. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House, p. 157.
- [6] Aho, A., Hopcroft, J., and Ullman, J. 1974. *The Design and Analysis of Computer Programs*. Reading, MA: Addison-Wesley, p. 2.
- [7] Buhr, R. August 22, 1988. *Machine Charts for Visual Prototyping in System Design*. SCE Report 88-2. Ottawa, Canada: Carleton University.

Глава 6. Процесс

- [1] Druke, M. 1989. Private communication.
- [2] Jones, C. September 1984. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, vol. SE-10 (5).
- [3] Curtis, B. May 17, 1989. . . . *But You Have to Understand, This Isn't the Way We Develop Software at Our Company*. MCC Technical Report Number STP-203-89. Austin, TX: Microelectronics and Computer Technology Corporation, p. x.
- [4] Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group, p. 290.
- [5] Boehm, B. August 1986. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, vol. 11 (4), p. 22.
- [6] Bailin, S. 1988. *Remarks on Object-Oriented Requirements Specification*. Laurel, MD: CTA Incorporated, p. 1.
- [7] Brownsword, L. 1989. Private communication.
- [8] Beck, K., and Cunningham, W. October 1989. A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices*, vol. 24 (10).

Глава 7. Традиционные методы

- [1] Dijkstra, E. May 1968. The Structure of the "THE" Multiprogramming System. *Communications of the ACM*, vol. 11 (5), p. 341.
- [2] Kishida, K., Teramoto, M., Torri, K., and Urano, Y. September 1988. Quality Assurance Technology in Japan. *IEEE Software*, vol. 4 (5), p. 13.
- [3] Hawryszkiewicz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates, p. 115.
- [4] Boehm, B. August 1986. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, vol. 11 (4), p. 23.
- [5] Hatley, D., and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House, p. 27.
- [6] Mellor, S., Hecht, A., Tryon, D., and Hywari, W. September 1988. Object-Oriented Analysis: Theory and Practice, Course Notes, in *Object-Oriented Programming Systems, Languages, and Applications*. San Diego, CA: OOPSLA'88, p. 1.3.
- [7] Yourdon, E. 1989. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [8] DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.
- [9] Gane, C., and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [10] Ward, P., and Mellor, S. 1985. *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Prentice-Hall.
- [11] Hatley and Pirbhai. *Strategies for Real-Time*.
- [12] DeMarco, T. 1987. Private communication.
- [13] Shlaer, S., and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press.
- [14] Coad, P. Summer 1989. OOA: Object-Oriented Analysis. *American Programmer*, vol. 2 (7-8).
- [15] Smith, M., and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects*. Seattle, WA: Boeing Commercial Airplane Support Division, p. 133.
- [16] Marca, D., and McGowan, C. 1988. *SADT™: Structured Analysis and Design Technique*. New York, NY: McGraw-Hill.
- [17] Alford, M. 1983. Derivation of Element-Relation-Attribute Database Requirements by Decomposition of System Functions, in *Entity-Relationship Approach to Software Engineering*. Amsterdam, The Netherlands: Elsevier Science Publishers.
- [18] Stoecklin, S., Adams, E., and Smith, S. 1987. *Object-Oriented Analysis*. Tallahassee, FL: East Tennessee State University.
- [19] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press. pp. 261-265.
- [20] Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations, *AI Magazine*, vol. 6 (4), p. 41.
- [21] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall, p. 340.

- [22] As quoted in Sommerville, I. 1989. *Software Engineering*. 3rd ed. Wokingham, England: Addison-Wesley, p. 546.
- [23] As quoted in Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys*, vol. 10 (2), p. 204.
- [24] Showalter, J. 1989. Private communication.
- [25] Davis, A., Bersoff, E., and Comer, E. October 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering*, vol. 14 (10), p. 1456.
- [26] Lang, K., and Peraltmutter, B. November 1986. Oaklisp: an Object-Oriented Scheme with First-Class Types. *SIGPLAN Notices*, vol. 21 (11), p. 34.
- [27] Meyrowitz, N. November 1986. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. *SIGPLAN Notices*, vol. 21 (11), p. 200.
- [28] Schmucker, K. 1986. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden, p. 11.
- [29] Simonian, R., and Crone, M. November/December 1988. InnovAda: True Object-Oriented Programming in Ada. *Journal of Object-Oriented Programming*, vol. 1 (4), p. 19.
- [30] Pascoe, G. August 1986. Elements of Object-Oriented Programming. *Byte*, vol. 11 (8), p. 144.
- [31] Russo, V., and Kaplan, S. 1988. A C++ Interpreter for Scheme. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association, p. 106.
- [32] Kempf, R. October 1987. Teaching Object-Oriented Programming with the KEE System. *SIGPLAN Notices*, vol. 22 (12), p. 11.

Часть III. ПРИМЕНЕНИЯ

Minsky, M. April 1970. Form and Content in Computer Science. *Journal of the Association for Computing Machinery*, vol. 17 (2), p. 197.

Глава 8. Smalltalk. Система домашнего отопления

- [1] White, S. October 1986. Panel Problem: Software Controller for an Oil Hot Water Heating System. *Proceedings of COMPSAC*. New York, NY: Computer Society Press of the IEEE. pp. 276-277.
- [2] Kerth, N. Private communication.
- [3] *Pluggable Gauges Version 1.0 User Manual*. 1987. Cary, NC: Knowledge Systems.

Глава 9. Object Pascal. Инструментальное средство разработки конструкций геометрической оптики

- [1] Sears, F., Zemansky, M., and Young, H. 1987. *University Physics*. 7th ed. Reading, MA: Addison-Wesley, p. 887.
- [2] Sears, Zemansky, and Young. *University Physics*, p. 890.
- [3] *Inside Macintosh*, Vol. 1-5. 1988. Reading, MA: Addison-Wesley.
- [4] Scheiffler, R., and Gettys, J. 1986. The X Window System. *ACM Transactions on Graphics*, vol. 63.
- [5] *Open Look Graphical User Interface Functional Specification*. 1990. Reading, MA: Addison-Wesley.
- [6] Durant, D., Carlson, G., and Yao, P. 1987. *Programmer's Guide to Windows*. Berkeley, CA: Sybex.
- [7] *IBM Operating System/2 Seminar Proceedings, IBM OS/2 Standard Edition Version 1.1, IBM Operating System/2 Update, Presentation Manager*. April 1988. Boca Raton, FL: International Business Machines.
- [8] *MacApp: The Expandable Macintosh Application*, version 2.0B9. 1989. Cupertino, CA: Apple Computer.
- [9] *MacApp*, p. 2.
- [10] *Inside Macintosh*.
- [11] *OSF/Motif Style Guide, Version 1.0*. 1989. Cambridge, MA: Open Software Foundation.

Глава 10. C++. Система регистрации ошибок в программных средствах

- [1] Levy, P. 1989. Private communication.
- [2] Date, C. 1981. *An Introduction to Database Systems*. Vol. 1. Reading, MA: Addison-Wesley, p. 4.
- [3] Date. *An Introduction*, p. 10.
- [4] Hawryszkiewicz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates, p. 425.
- [5] Wiorowski, G., and Kull, D. 1988. *DB2 Design and Development Guide*. Reading, MA: Addison-Wesley, p. 29.
- [6] Date. *An Introduction*, p. 63.
- [7] Wiorowski and Kull. *DB2 Design*, p. 2.
- [8] Date. *An Introduction*. p. 237.
- [9] Date. *An Introduction*. p. 238.
- [10] Wiorowski and Kull. *DB2 Design*. p. 15.
- [11] Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, p. 461.
- [12] *Oracle for Macintosh: References, Version 1.1*. 1989. Belmont, CA: Oracle, p. 5-118.
- [13] Hawryszkiewicz. *Database Analysis*. p. 431.

Глава 11. Common Lisp Object System. Система дешифрования

- [1] Meyer, C., and Matyas. 1982. *Cryptography*. New York, NY: John Wiley and Sons, p. 1.
- [2] Enman, L., Lark, J., and Hayes-Roth, F. December 1988. ABE: An Environment for Engineering Intelligent Systems. *IEEE Transactions on Software Engineering*, vol. 14 (12), p. 1758.
- [3] Nii, P. Summer 1986. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine*, vol. 7 (2), p. 46.
- [4] Englemore, R., and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley, p. 16.
- [5] Ibid., p. 19.
- [6] Ibid., p. 6.
- [7] Ibid., p. 12.
- [8] Nii. Blackboard Systems, p. 43.
- [9] Englemore and Morgan. *Blackboard Systems*, p. 11.

Глава 12. Ada. Система управления движением

- [1] *Rockwell Advanced Railroad Electronic Systems*. 1989. Cedar Rapids, IA: Rockwell International.
- [2] Murphy, E. December 1988. All Aboard for Solid State. *IEEE Spectrum*, vol. 25 (13), p. 42.
- [3] Murphy. All Aboard.
- [4] Tanenbaum, A. 1981. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall.

Заключение

Lefrancois, G. 1977. *Of Children: An Introduction to Child Development, Second Edition*. Belmont, CA: Wadsworth, p. 371.

Приложения

- [1] Wulf, W. January 1980. Trends in the Design and Implementation of Programming Languages. *IEEE Computer*, vol. 13 (1), p. 15.
- [2] Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur.
- [3] Schmucker, K. 1986. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden, p. 346.
- [4] Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, p. 9.

- [5] Borning, A., and Ingalls, D. 1982. Multiple Inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI.
- [6] Goldberg, A., and Robson, D. 1989. *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley.
- [7] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.
- [8] Krasner, G. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley.
- [9] Schmucker, K. August 1986. Object-Oriented Languages for the Macintosh. *Byte*, vol. 11 (8), p. 179.
- [10] *Macintosh Programmer's Workshop Pascal 3.0 Reference*. 1989. Cupertino, CA: Apple Computer.
- [11] Stroustrup, B. 1986. *The C++ Programming Language*. Reading, MA: Addison-Wesley, p. 4. .
- [12] Gorlen, K. 1989. An Introduction to C++, in *UNIX System V AT&T C++ Language System, Release 2.0 Selected Readings*. Murray Hill, NJ: AT&T Bell Laboratories, p. 2-1.
- [13] *UNIX System V AT&T C++ Language System, Release 2.0 Product Reference Manual*. 1989. Murray Hill, NJ: AT&T Bell Laboratories.
- [14] *UNIX System V AT&T C++ Language System, Release 2.0 Selected Readings*. 1989. Murray Hill, NJ: AT&T Bell Laboratories.
- [15] *UNIX System V AT&T C++ Language System, Release 2.0 Release Notes*. 1989. Murray Hill, NJ: AT&T Bell Laboratories.
- [16] *UNIX System V AT&T C++ Language System, Release 2.0 Library Manual*. 1989. Murray Hill, NJ: AT&T Bell Laboratories.
- [17] Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley, p. 215.
- [18] Bobrow, D. 1990. Private communication.
- [19] Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D. September 1988. Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Notices*, vol. 23.
- [20] *Reference Manual for the Ada Programming Language*. February 1983. Washington, D.C.: Department of Defense, Ada Joint Program Office, p. 1-3.
- [21] Ibid.
- [22] Saunders, J. March/April 1989. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, vol. 1 (6).
- [23] Ibid., p. 6.

Библиография

Библиография разделена на одиннадцать разделов, проименованных латинскими буквами от А до К. Ссылки на источники в конце каждой главы имеют следующую форму [метка раздела, <род>]. Например, ссылка Brooks [Н, 1975] относится к книге *The Mythical Man-Month* в разделе Н («Разработка программных продуктов») библиографии.

А. Классификация

- Aquinas, T. *Summa Theologica*. Vol. 19 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica.
- Aristotle. *Categories*. Vol. 8 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica.
- Classification Society of North America. *Journal of Classification*. New York, NY: Springer-Verlag.
- Coombs, C., Raiffa, H., and Thrall, R. 1954. Some Views on Mathematical Models and Measurement Theory. *Psychological Review* vol. 61 (2).
- Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28 (6).
- Darwin, C. *The Origin of Species*. Vol. 49 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica.
- Descartes, R. *Rules for the Direction of the Mind*. Vol. 31 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica.
- Flood, R., and Carson, E. 1988. *Dealing with Complexity*. New York, NY: Plenum Press.
- Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. Chicago, IL: The University of Chicago Press.
- Le-francois, G. 1977. *Of Children: An Introduction to Child Development*. 2nd ed. Belmont, CA: Wadsworth.
- Lewin, R. 4 November 1988. Family Relationships Are a Biological Conundrum. *Science* vol. 242.
- Maier, H. 1969. *Three Theories of Child Development: The Contributions of Erik H. Erickson, Jean Piaget, and Robert R. Sears, and Their Applications*. New York, NY: Harper and Row.
- May, R. 16 September 1988. How Many Species Are There on Earth? *Science* vol. 241.
- Michalski, R., and Stepp, R. 1983. Learning from Observation: Conceptual Clustering, in *Machine Learning: An Artificial Intelligence Approach*. ed. R. Michalski, J. Carbonell, and T. Mitchell. Palo Alto, CA: Tioga.
- Miller, G. March 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review* vol. 63 (2).
- Minsky, M. April 1970. Form and Content in Computer Science. *Journal of the Association for Computing Machinery* vol. 17 (2).

- Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster.
- Moldovan, D., and Wu, C. December 1988. A Hierarchical Knowledge-Based System for Airplane Classification. *IEEE Transactions on Software Engineering* vol. 14 (12).
- Newell, A., and Simon, H. 1972. *Human Problem Solving*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY: Basic Books.
- Plato. *Statesman*. Vol. 7 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica.
- Siegler, R., and Richards, D. 1982. The Development of Intelligence, in *Handbook of Human Intelligence*. ed. R. Sternberg. Cambridge, London: Cambridge University Press.
- Simon, H. 1982. *The Sciences of the Artificial*. Cambridge, MA: The MIT Press.
- Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.
- Stepp, R., and Michalski, R. 1986. Conceptual Clustering of Structured Objects: A Goal-Oriented Approach. *Artificial Intelligence* vol. 28 (1).
- Stevens, S. June 1946. On the Theory of Scales of Measurement, *Science* vol. 103 (2684).
- Stillings, N., Feinstein, M., Garfield, J., Rissland, E., Rosenbaum, D., Weisler, S., and Baker-Ward, L. 1987. *Cognitive Science: An Introduction*. Cambridge, MA: The MIT Press.

В. Объектно-ориентированный анализ

- Arango, G. May 1989. Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes* vol. 14 (3).
- Bailin, S. 1988. *Remarks on Object-Oriented Requirements Specification*. Laurel, MD: Computer Technology Associates.
- Bailin, S., and Moore, J. 1987. *An Object-Oriented Specification Method for Ada*. Laurel, MD: Computer Technology Associates.
- Borgida, A., Mylogoulos, J., and Wong, H. 1984. Generalization/Specialization as a Basis for Software Specification, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Cernosek, G., Monterio, E., and Pribyl, W. 1987. *An Entity-Relationship Approach to Software Requirements Analysis for Object-Based Development*. Houston, TX: McDonnell Douglas Astronautics.
- Coad, P. Summer 1989. OOA: Object-Oriented Analysis. *American Programmer* vol. 2 (7-8).
- Coad, P., and Yourdon, E. 1990. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- Dahl, O.-J. 1987. Object-Oriented Specifications, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.

- DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.
- EV8 Software Engineering. 1989. *Object-Oriented Requirements Analysis*. Frederick, MD.
- Gane, C., and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- Hatley, D., and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.
- Iscoe, N. 1988. *Domain Models for Program Specification and Generation*. Austin, TX: University of Texas.
- Iscoe, N., Browne, J., and Werth, J. 1989. *Modeling Domain Knowledge: An Object-Oriented Approach to Program Specification and Generation*. Austin, TX: The University of Texas.
- Marca, D., and McGowan, C. 1988. *SADT™: Structured Analysis and Design Technique*. New York, NY: McGraw-Hill.
- McMenamin, S., and Palmer, J. 1984. *Essential Systems Analysis*. New York, NY: Yourdon Press.
- Mellor, S., Hecht, A., Tryon, D., and Hywar, W. September 1988. Object-Oriented Analysis: Theory and Practice, Course Notes, from *Object-Oriented Programming Systems, Languages, and Applications*. San Diego, CA: OOPSLA'88.
- Moore, J., and Bailin, S. 1988. *Position Paper on Domain Analysis*. Laurel, MD: Computer Technology Associates.
- Page-Jones, M., and Weiss, S. Summer 1989. Synthesis: An Object-Oriented Analysis and Design Method. *American Programmer* vol. 2 (7-8).
- Saeki, M., Horai, H., and Enomoto, H. May 1989. Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- Shemer, I. June 1987. Systems Analysis: A Systemic Analysis of a Conceptual Model. *Communications of the ACM* vol. 30 (6).
- Shlaer, S., and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press.
- Shlaer, S., and Mellor, S. July 1989. An Object-Oriented Approach to Domain Analysis. *Software Engineering Notes* vol. 14 (5).
- Shlaer, S., and Mellor, S. Summer 1989. Understanding Object-Oriented Analysis. *American Programmer* vol. 2 (7-8).
- Stoecklin, S., Adams, E., and Smith, S. 1987. *Object-Oriented Analysis*. Tallahassee, FL: East Tennessee State University.
- Sully, P. Summer 1989. Structured Analysis: Scaffolding for Object-Oriented Development. *American Programmer* vol. 2 (7-8).
- Tsai, J., and Ridge, J. November 1988. Intelligent Support for Specifications Transformation. *IEEE Software* vol. 5 (6).
- Veryard, R. 1984. *Pragmatic Data Analysis*. Oxford, England: Blackwell Scientific Publications.
- Ward, P. March 1989. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software* vol. 6 (2).

Weinberg, G. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House.

С. Примеры объектно-ориентированного применения

- Abdali, K., Cherry, G., and Soiffer, N. November 1986. A Smalltalk System for Algebraic Manipulation. *SIGPLAN Notices* vol. 21 (11).
- Almes, G., and Holman, C. September 1987. Edmas: An Object-Oriented, Locally Distributed Mail System. *IEEE Transactions on Software Engineering* vol. SE-13 (9).
- Anderson, D. November 1986. Experience with Flamingo: A Distributed, Object-Oriented User Interface System. *SIGPLAN Notices* vol. 21 (11).
- Archer, J., and Devlin, M. 1987. *Rational's Experience Using Ada for Very Large Systems*. Mountain View, CA: Rational.
- Bagrodia, R., Chandy, M., and Misra, J. June 1987. A Message-Based Approach to Discrete-Event Simulation. *IEEE Transactions on Software Engineering* vol. SE-13 (6).
- Barry, B. October 1989. Prototyping a Real-Time Embedded System in Smalltalk. *SIGPLAN Notices* vol. 24 (10).
- Barry, B., Altoft, J., Thomas, D., and Wilson, M. October 1987. Using Objects to Design and Build Radar ESM Systems. *SIGPLAN Notices* vol. 22 (12).
- Bezivin, J. October 1987. Some Experiments in Object-Oriented Simulation. *SIGPLAN Notices* vol. 22 (12).
- Bhaskar, K., and Peckol, J. November 1986. Virtual Instruments: Object-Oriented Program Synthesis. *SIGPLAN Notices* vol. 21 (11).
- Bjornstedt, A., and Britts, S. September 1988. AVANCE: An Object Management System. *SIGPLAN Notices* vol. 23 (11).
- Bobrow, D., and Stefik, M. February 1986. Perspectives on Artificial Intelligence Programming. *Science* vol. 231.
- Boltuck-Pasquier, J., Grossman, E., and Collaud, G. August 1988. Prototyping an Interactive Electronic Book System Using an Object-Oriented Approach. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Bonar, J., Cunningham R., and Schultz, J. November 1986. An Object-Oriented Architecture of Intelligent Tutoring Systems. *SIGPLAN Notices* vol. 21 (11).
- Booch, G. 1987. *Software Components with Ada: Structures, Tools, and Subsystems*. Menlo Park, CA: Benjamin/Cummings.
- Borning, A. October 1981. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems* vol. 3 (4).
- Bowman, W., and Flegel, B. August 1981. ToolBox: A Smalltalk Illustration System. *Byte* vol. 6 (8).
- Britcher, R., and Craig, J. May 1986. Using Modern Design Practices to Upgrade Aging Software Systems. *IEEE Software* vol. 3 (3).
- Britton, K., and Parnas, D. December 8, 1981. *A-7E Software Module Guide*, Report 4702. Washington, D.C.: Naval Research Laboratory.

- Bruck, D. 1988. Modeling of Control Systems with C++ and PHIGS. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Budd, T. January 1989. The Design of an Object-Oriented Command Interpreter. *Software – Practice and Experience* vol. 19 (1).
- Call, L., Cohrs, D., and Miller, B. October 1987. CLAM – an Open System for Graphical User Interfaces. *SIGPLAN Notices* vol. 22 (12).
- Caplinger, M. October 1987. An Information System Based on Distributed Objects. *SIGPLAN Notices* vol. 22 (12).
- Cargill, T. November 1986. Pi: A Case Study in Object-Oriented Programming. *SIGPLAN Notices* vol. 21 (11).
- Cmelik, R., and Genani, N. May 1988. Dimensional Analysis with C++. *IEEE Software* vol. 5 (3).
- Cointe, P., Briot, J., and Serpette, B. 1987. The Formes System: A Musical Application of Object-Oriented Concurrent Programming, in *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Coutaz, J. September 1985. Abstractions for User Interface Design. *IEEE Computer* vol. 18 (9).
- Dasgupta, P. November 1986. A Probe-Based Monitoring Scheme for an Object-Oriented Operating System. *SIGPLAN Notices* vol. 21 (11).
- Davidson, C., and Moseley, R. 1987. *An Object-Oriented Real-Time Knowledge-Based System*. Albuquerque, NM: Applied Methods.
- Dietrich, W., Nackman, L., and Gracer, F. October 1989. Saving a Legacy with Objects. *SIGPLAN Notices* vol. 24 (10).
- Dijkstra, E. May 1968. The Structure of the "THE" Multiprogramming System. *Communications of the ACM* vol. 11 (5).
- Durand, G., Benkiran, A., Durel, C., Nga, H., and Tag, M. 9 March 1988. *Distributed Mail Service in CSE System*. Paris, France: Synergie Informatique et Development.
- Englemore, R., and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley.
- Epstein, D., and LaLonde, W. September 1988. A Smalltalk Window System Based on Constraints. *SIGPLAN Notices* vol. 23 (11).
- Ewing, J. November 1986. An Object-Oriented Operating System Interface. *SIGPLAN Notices* vol. 21 (11).
- Fenton, J., and Beck, K. October 1989. Playground: An Object-Oriented Simulation System with Agent Rules for Children of All Ages. *SIGPLAN Notices* vol. 24 (10).
- Fischer, G. 1987. *An Object-Oriented Construction and Tool Kit for Human-Computer Communication*. Boulder, CO: University of Colorado Department of Computer Science and Institute of Cognitive Science.
- Foley, J., and van Dam, A. 1982. *Fundamentals of Interactive Computer Graphics*. Reading, MA: Addison-Wesley.
- Frankowski, E. 20 March 1986. *Advantages of the Object Paradigm for Prototyping*. Golden Valley, MN: Honeywell.
- Freburger, K. October 1987. RAPID: Prototyping Control Panel Interfaces. *SIGPLAN Notices* vol. 22 (12).

- Funk, D. 1986. Applying Ada to Beech Starship Avionics. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Garrett, N., and Smith, K. November 1986. Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application. *SIGPLAN Notices* vol. 21 (11).
- Goldberg, A. 1978. *Smalltalk in the Classroom*. Palo Alto, CA: Xerox Palo Alto Research Center.
- Gorlen, K. December 1987. An Object-Oriented Class Library for C++ Programs. *Software - Practice and Experience* vol. 17 (12).
- Gray, L. 1987. *Transferring Object-Oriented Design Techniques into Use. AWIS Experience*. Fairfax, VA: TRW Federal Systems Group.
- Grimshaw, A., and Liu, J. October 1987. Mentat: An Object-Oriented Macro Data Flow System. *SIGPLAN Notices* vol. 22 (12).
- Grossman, M., and Ege, R. October 1987. Logical Composition of Object-Oriented Interfaces. *SIGPLAN Notices* vol. 22 (12).
- Gutfreund, S. October 1987. Maniplcons in ThinkerToy. *SIGPLAN Notices* vol. 22 (12).
- Harrison, W., Shilling, J., and Sweeney, P. October 1989. Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm. *SIGPLAN Notices* vol. 24 (10).
- Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K. September 1988. Fabrik: A Visual Programming Environment. *SIGPLAN Notices* vol. 23 (11).
- Jacky, J., and Kalet, I. November 1986. An Object-Oriented Approach to a Large Scientific Application. *SIGPLAN Notices* vol. 21 (11).
- Jerrell, M. October 1989. Function Minimization and Automatic Differentiation using C++. *SIGPLAN Notices* vol. 24 (10).
- Johnson, R., and Foote, B. June/July 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming* vol. 1 (2).
- Jones, M., and Rashid, R. November 1986. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. *SIGPLAN Notices* vol. 21 (11).
- Kay, A., and Goldberg, A. March 1977. Personal Dynamic Media. *IEEE Computer*.
- Kerr, R., and Percival, D. October 1987. Use of Object-Oriented Programming in a Time Series Analysis System. *SIGPLAN Notices* vol. 22 (12).
- Kuhl, F. 1988. *Object-Oriented Design for a Workstation for Air Traffic Control*. McLean, VA: The MITRE Corporation.
- LaPolla, M. 1988. *On the Classification of Object-Oriented Design: The Object-Oriented Design of the AirLand Battle Management Menu System*. Austin, TX: Lockheed Software Technology Center.
- Lea, D. 12 August 1988. *User's Guide to GNU C++ Library*. Cambridge, MA: Free Software Foundation.
- Lea, D. 1988. The GNU C++ Library. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Leadbetter, L., and Cox, B. June 1985. Software-ICs. *Byte* vol. 10 (6).
- Lee, K., and Rissman, M. February 1989. *An Object-Oriented Solution Example: A Flight Simulator Electrical System*. Pittsburgh, PA: Software Engineering Institute.

- Lee, K., Rissman, M., D'Ippolito, R., Plinta, C., and Van Scoy, R. December 1987. *An OOL Paradigm for Flight Simulators*, Report CMU/SEI-87-TR-43. Pittsburgh, PA: Software Engineering Institute.
- Levy, P. 1987. *Implementing Systems Software in Ada*. Mountain View, CA: Rational.
- Linton, M., Vlissides, J., and Calder, P. February 1989. Composing User Interfaces with InterViews. *IEEE Computer*. vol. 22 (2).
- Liu, L., and Horowitz, E. February 1989. Object Database Support for a Software Project Management Environment. *SIGPLAN Notices* vol. 24 (2).
- Locke, D., and Goodenough, J. 1988. *A Practical Application of the Ceiling Protocol in a Real-Time System*, Report CMU/SEI-88-SR-3. Pittsburgh, PA: Software Engineering Institute.
- Madany, P., Leyens, D., Russo, V., and Campbell, R. 1988. A C++ Class Hierarchy for Building UNIX-like File Systems. *Proceedings of USENIX C++ Conference* Berkeley, CA: USENIX Association.
- Madduri, H., Raeuchle, T., and Silverman, J. 1987. *Object-Oriented Programming for Fault-Tolerant Distributed Systems*. Golden Valley, MN: Honeywell Computer Science Center.
- Maloney, J., Borning, A., and Freeman-Benson, B. October 1989. Constraint Technology for User Interface Construction in ThingLab II. *SIGPLAN Notices* vol. 24 (10).
- McDonald, J. October 1989. Object-Oriented Programming for Linear Algebra. *SIGPLAN Notices* vol. 24 (10).
- Meyrowitz, N. November 1986. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. *SIGPLAN Notices* vol. 21 (11).
- Miller, M., Cunningham, H., Lee, C., and Vegdahl, S. November 1986. The Application Accelerator Illustration System. *SIGPLAN Notices* vol. 21 (11).
- Mohan, L., and Kashyap, R. May 1988. An Object-Oriented Knowledge Representation for Spatial Information. *IEEE Transactions on Software Engineering* vol. 14 (5).
- Mraz, R. December 1986. *Performance Evaluation of Parallel Branch and Bound Search with the Intel iPSE Hypercube Computer*. Wright-Patterson Air Force Base, Ohio: Air Force Institute of Technology.
- Muller, H., Rose, J., Kempf, J., and Stansbury, T. October 1989. The Use of Multimethods and Method Combination in a CLOS-Based Window Interface. *SIGPLAN Notices* vol. 24 (10).
- Murphy, E. December 1988. All Aboard for Solid State. *IEEE Spectrum* vol. 25 (13).
- NeXT Embraces a New Way of Programming. 25 November 1988. *Science* vol. 242.
- Orden, E. 1987. Application Talk. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1 (1). Everett, WA: Object-Oriented Programming for Smalltalk Application Developers Association.
- Oshima, M., and Shirai, Y. July 1983. Object Recognition Using Three-Dimensional Information. *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. 5 (4).
- Page, T., Berson, S., Cheng, W., and Muntz, R. October 1989. An Object-Oriented Modeling Environment. *SIGPLAN Notices* vol. 24 (10).

- Pashtan, A. 1982. Object-Oriented Operating Systems: An Emerging Design Methodology. *Proceedings of the ACM '82 Conference*. New York, NY: Association of Computing Machinery.
- Piersol, K. November 1986. Object-Oriented Spreadsheets: The Analytic Spreadsheet Package. *SIGPLAN Notices* vol. 21 (11).
- Plinta, C., Lee, K., and Rissman, M. 29 March 1989. A Model Solution for C3I: Message Translation and Validation. Pittsburgh, PA: Software Engineering Institute.
- Pope, S. April/May 1988. Building Smalltalk-80-based Computer Music Tools. *Journal of Object-Oriented Programming* vol. 1 (1).
- Rockwell International. 1989. *Rockwell Advanced Railroad Electronic Systems*. Cedar Rapids, IA.
- Rubin, K., Jones, P., Mitchell, C., and Goldstein, T. September 1988. A Smalltalk Implementation of an Intelligent Operator's Associate. *SIGPLAN Notices* vol. 23 (11).
- Ruspini, E., and Fraley, R. 1983. ID: An Intelligent Information Dictionary System, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Russo, V., Johnston, G., and Campbell, R. September 1988. Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 23 (11).
- Sampson, J., and Womble, B. 1988. *SEND: Simulation Environment for Network Design*. Dallas, TX: Southern Methodist University.
- Scaletti, C., and Johnson, R. September 1988. An Interactive Environment for Object-Oriented Music Composition and Sound Synthesis. *SIGPLAN Notices* vol. 23 (11).
- Schoen, E., Smith, R., and Buchanan, B. December 1988. Design of Knowledge-Based Systems with a Knowledge-Based Assistant. *IEEE Transactions on Software Engineering* vol. 14 (12).
- Schulert, A., and Erf, K. 1988. Open Dialogue: Using an Extensible Retained Object Workspace to Support a UIMS. *Proceedings of USENIX C++ Conference* Berkeley, CA: USENIX Association.
- Scott, R., Reddy, P., Edwards, R., and Campbell, D. 1988. GPIO: Extensible Objects for Electronic Design. *Proceedings of USENIX C++ Conference* Berkeley, CA: USENIX Association.
- Smith, R., Barth, P., and Young, R. 1987. A Substrate for Object-Oriented Interface Design. *Research Directions in Object-Oriented Programming* Cambridge, MA: The MIT Press.
- Smith, R., Dinitz, R., and Barth, P. November 1986. Impulse-86: A Substrate for Object-Oriented Interface Design. *SIGPLAN Notices* vol. 21 (11).
- Sneed, H., and Gawron, W. 1983. The Use of the Entity/Relationship Model as a Schema for Organizing the Data Processing Activities at the Bavarian Motor Works, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Snodgrass, R. 1987. An Object-Oriented Command Language, in *Object-Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Sridhar, S. September 1988. Configuring Stand-Alone Smalltalk-80 Applications. *SIGPLAN Notices* vol. 23 (11).

- Stokes, R. 1988. Prototyping Database Applications with a Hybrid of C++ and 4GL. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Szur, M., and Miller, P. September 1988. Transportable Applications Environment (TAE) PLUS: Experiences in "Object"ively Modernizing a User Interface Environment. *SIGPLAN Notices* vol. 23 (11).
- Szekely, P., and Myers, B. September 1988. A User Interface Toolkit Based on Graphical Objects and Constraints. *SIGPLAN Notices* vol. 23 (11).
- Tanner, J. 1 April 1986. *Fault Tree Analysis in an Object-Oriented Environment*. Mountain View, CA: IntelliCorp.
- Temte, M. November/December 1984. Object-Oriented Design and Ballistics Software. *Ada Letters* vol. 4 (3).
- Tripathi, A., and Aksit, M. November/December 1988. Communication, Scheduling, and Resource Management in SINA. *Journal of Object-Oriented Programming* vol. 1 (4).
- Tripathi, A., Ghonami, A., and Schmitz, T. 1987. Object Management in the NEXUS Distributed Operating System. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.
- Ursprung, P., and Zehnder, C. 1983. HIQUEL: An Interactive Query Language to Define and Use Hierarchies, in *Entity-relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- van der Meulen, P. October 1987. INSIST: Interactive Simulation in Smalltalk. *SIGPLAN Notices* vol. 22 (12).
- Vernon, V. September/October 1989. The Forest for the Trees. *Programmer's Journal* vol. 7 (5).
- Vines, D., and King, T. 1987. *Experiences in Building a Prototype Object-Oriented Framework in Ada*. Minneapolis, MN: Honeywell.
- Vlissides, J., and Linton, M. 1988. Applying Object-Oriented Design to Structured Graphics. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Volz, R. Mudge, T., and Gal, D. 1987. Using Ada as a Programming Language for Robot-Based Manufacturing Cells, in *Object-Oriented Computing: Concepts* vol. 1, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Walther, S., and Peskin, R. October 1989. Strategies for Scientific Prototyping in Smalltalk. *SIGPLAN Notices* vol. 24 (10).
- Weinand, A., Gamma, E., and Marty, R. September 1988. ET++ - An Object-Oriented Application Framework in C++. *SIGPLAN Notices* vol. 23 (11).
- White, S. October 1986. Panel Problem: Software Controller for an Oil Hot Water Heating System. *Proceedings of COMPSAC*. New York, NY: Computer Society Press of the IEEE.
- Wirfs-Brock, R. September 1988. An Integrated Color Smalltalk-80 System. *SIGPLAN Notices* vol. 23 (11).
- Yoshida, N., and Hino, K. September 1988. An Object-Oriented Framework of Pattern Recognition. *SIGPLAN Notices* vol. 23 (11).
- Yoshida, T., and Tokoro, M. 31 March 1986. *Distributed Queueing Network Simulation: An Application of a Concurrent Object-Oriented Language*. Yokohama, Japan: Keio University.

Young, R. October 1987. An Object-Oriented Framework for Interactive Data Graphics. *SIGPLAN Notices* vol. 22 (12).

D. Архитектура объектно-ориентированных разработок

Athas, W., and Seitz, C. August 1988. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* vol. 21 (8).

Dahlby, S., Henry, G., Reynolds, D., and Taylor, P. 1982. The IBM System/38: A High Level Machine, in *Computer Structures: Principles and Examples*, ed. G. Bell and A. Newell. New York, NY: McGraw-Hill.

Dally, W., and Kajiya, J. March 1985. An Object-Oriented Architecture. *SIGARCH Newsletter* vol. 13 (3).

Fabry, R. 1987. Capability-Based Addressing, in *Object-Oriented Computing: Implementations* vol. 2, ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.

Flynn, M. October 1980. Directions and Issues in Architecture and Language. *IEEE Computer* vol. 13 (10).

Harland, D., and Beloff, B. December 1986. Microcoding an Object-Oriented Instruction Set. *Computer Architecture News* vol. 14 (5).

Iliffe, J. 1982. *Advanced Computer Design*. London, England: Prentice/Hall International.

Intel. 1981. *iAPX 432 Object Primer*. Santa Clara, CA.

Ishikawa, Y., and Tokoro, M. March 1984. The Design of an Object-Oriented Architecture. *SIGARCH Newsletter* vol. 12 (3).

Kavi, K., and Chen, D. 1987. Architectural Support for Object-Oriented Languages. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.

Lahtinen, P. September/October 1982. A Machine Architecture for Ada. *Ada Letters* vol. 2 (2).

Lampson, B., and Pier, K. January 1981. A Processor for a High-Performance Personal Computer, in *The Dorado: A High Performance Personal Computer*, Report CSL-81-1. Palo Alto, CA: Xerox Palo Alto Research Center.

Langdon, G. 1982. *Computer Design*. San Jose, CA: Computeach Press.

Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press.

Lewis, D., Galloway, D., Francis, R., and Thomson, B. November 1986. Swamp: A Fast Processor for Smalltalk-80. *SIGPLAN Notices* vol. 21 (11).

Mashburn, H. 1982. The C.mmp/Hydra Project: An Architectural Overview, in *Computer Structures: Principles and Examples*, ed. G. Bell and A. Newell. New York, NY: McGraw-Hill.

Myers, G. 1982. *Advances in Computer Architecture*, 2nd ed. New York, NY: John Wiley and Sons.

Rattner, J. 1982. Hardware/Software Cooperation in the iAPX-432. *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*. New York, NY: Association of Computing Machinery.

Rose, J. September 1988. Fast Dispatch Mechanisms for Stock Hardware. *SIGPLAN Notices* vol. 23 (11).

- Samples, D., Ungar, D., and Hilfinger, P. November 1986. SOAR: Smalltalk Without Bytecodes. *SIGPLAN Notices* vol. 21 (11).
- Soltis, R., and Hoffman, R. 1987. Design Considerations for the IBM System/38, in *Object-Oriented Computing: Implementations* vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Thacker, C., McCreight, E., Lampson, B., Sproull, R., and Boggs, D. August 1979. *Alto: A Personal Computer*, Report CSL-79-11. Palo Alto, CA: Xerox Palo Alto Research Center.
- Ungar, D. 1987. *The Design and Evaluation of a High-Performance Smalltalk System*. Cambridge, MA: The MIT Press.
- Ungar, D., Blau, R., Foley, P., Samples, D., and Patterson, D. March 1984. Architecture of SOAR: Smalltalk on a RISC. *SIGARCH Newsletter* vol. 12 (3).
- Ungar, D., and Patterson, D. January 1987. What Price Smalltalk? *IEEE Computer* vol. 20 (1).
- Wah, B., and Li, G. April 1986. Survey on Special Purpose Computer Architectures for AI. *SIGART Newsletter*, no. 96.
- Wulf, W. January 1980. Trends in the Design and Implementation of Programming Languages. *IEEE Computer* vol. 13 (1).
- Wulf, W., Levin, R., and Harbison, S. 1981. *HYDRA/C.mmp: An Experimental Computer System*. New York, NY: McGraw-Hill.

Е. Объектно-ориентированный СУБД

- Alford, M. 1983. Derivation of Element-Relation-Attribute Database Requirements by Decomposition of System Functions, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Atkinson, M., Bailey, P., Chisholm, K., Cockshott, P., and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal* vol. 26 (4).
- Atkinson, M., and Buneman, P. June 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys* vol. 19 (2).
- Atkinson, M., and Morrison, R. October 1985. Procedures as Persistent Data Objects. *ACM Transactions on Programming Languages and Systems* vol. 7 (4).
- Bachman, C. 1983. The Structuring Capabilities of the Molecular Data Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Batini, C., and Lenzerini, M. 1983. A Methodology for Data Schema Integration in the Entity-Relationship Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Beech, D. 1987. Groundwork for an Object Database Model, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Beech, D. September 1988. Intensional Concepts in an Object Database Model. *SIGPLAN Notices* vol. 23 (11).
- Bertino, E. 1983. Distributed Database Design Using the Entity-Relationship Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

- Blackwell, P., Jajodia, S., and Ng, P. 1983. A View of Database Management Systems as Abstract Data Types, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Bloom, T. October 1987. Issues in the Design of Object-Oriented Database Programming Languages. *SIGPLAN Notices* vol. 22 (12).
- Bohrow, D., Fogelsong, D., and Miller, M. 1987. Definition Groups: Making Sources into First-class Objects, in *Research Directions in Object-Oriented Programming* ed. B. Shriver and P. Wegner. Cambridge, MA: The MIT Press.
- Brathwaite, K. 1983. An Implementation of A Data Dictionary to Support Databases Designed Using the Entity-Relationship (E-R) Approach, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Breazeal, J., Blattner, M., and Burton, H. 28 March 1986. *Data Standardization Through the Use of Data Abstraction*. Livermore, CA: Lawrence Livermore National Laboratory.
- Brodie, M. 1984. On the Development of Data Models, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Brodie, M., and Ridjanovic, D. 1984. On the Design and Specification of Database Transactions. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Carlson, C., and Arora, A. 1983. UPM: A Formal Tool for Expressing Database Update Semantics, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Casanova, M. 1983. Designing Entity-Relationship Schemes for Conventional Information Systems, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Cattell, R. May 1983. *Design and Implementation of a Relationship-Entity-Datum Data Model*, Report CSL-83-4. Palo Alto, CA: Xerox Palo Alto Research Center.
- Chen, P. March 1976. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems* vol. 1 (1).
- Chen, P. 1983. ER – A Historical Perspective and Future Directions, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Claybrook, B., Claybrook, A., and Williams, J. January 1985. Defining Database Views as Data Abstractions. *IEEE Transactions on Software Engineering* vol. SE-11 (1).
- Date, C. 1981, 1983. *An Introduction to Database Systems*. Reading, MA: Addison-Wesley.
- Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley.
- Date, C. 1987. *The Guide to the SQL Standard*. Reading, MA: Addison-Wesley.
- D'Cunha, A., and Radhakrishnan, T. 1983. Applications of E-R Concepts to Data Administration, *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Duhl, J., and Damon, C. September 1988. A Performance Comparison of Object and Relational Databases Using the Sun Benchmark. *SIGPLAN Notices* vol. 23 (11).

- Harland, D., and Beloff, B. April 1987. OBJEKT – A Persistent Object Store with an Integrated Garbage Collector. *SIGPLAN Notices* vol. 22 (4).
- Hawryszkiewicz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates.
- Hull, R., and King, R. September 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys* vol. 19 (3).
- Jajodia, S., Ng, P., and Springsteel, F. 1983. On Universal and Representative Instances for Inconsistent Databases, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Ketabchi, M., and Berzins, V. January 1988. Mathematical Model of Composite Objects and Its Application for Organizing Engineering Databases. *IEEE Transactions on Software Engineering* vol. 14 (1).
- Ketabchi, M., and Wiens, R. 1987. Implementation of Persistent Multi-User Object-Oriented Systems. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.
- Kim, W., Ballou, N., Chou, H., Garza, J., Woelk, D., and Banerjee, J. September 1988. Integrating an Object-Oriented Programming System with a Database System. *SIGPLAN Notices* vol. 23 (11).
- Kim, W., Banerjee, J., Chou, H., Garza, J., and Woelk, D. October 1987. Composite Object Support in an Object-Oriented Database System. *SIGPLAN Notices* vol. 22 (12).
- Kim, W., and Lochovsky, K. 1989. *Object-Oriented Concepts, Databases, and Applications*. Reading, MA: Addison-Wesley.
- Laenens, E., and Vermeir, D. August 1988. An Overview of OOPS+, An Object-Oriented Database Programming Language. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Larson, J., and Dwyer, P. 1983. Defining External Schemas for an Entity-Relationship Database, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Maier, D., and Stein, J. 1987. Development and Implementation of an Object-Oriented DBMS, in *Research Directions in Object-Oriented Programming* ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Margrave, G., Lusk, E., and Overbeek, R. 1983. Tools for the Creation of IMS Database Designs from Entity-Relationship Diagrams, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Mark, L., and Poussopoulos, N. 1983. Integration of Data, Schema, and Meta-schema in the Context of Self-documenting Data Models, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Marti, R. 1983. Integrating Database and Program Descriptions using an ER Data Dictionary, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Morrow, T., and Laursen, J. October 1987. A Pragmatic System for Shared Persistent Objects. *SIGPLAN Notices* vol. 22 (12).

- Mitchell, J., and Wegbreit, B. 1977. Schemes: A High-Level Data Structuring Concept, in *Current Trends in Programming Methodology: Data Structuring* vol. 4. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Morrison, R., Atkinson, M., Brown, A., and Dearle, A. April 1988. Bindings in Persistent Programming Languages. *SIGPLAN Notices* vol. 23 (4).
- Moss, E., Herlihy, M., and Zdonik, S. September 1988. Object-Oriented Databases, Course Notes, from *Object-Oriented Programming Systems, Languages, and Applications*. San Diego, CA: OOPSLA'88.
- Nastos, M. January 1988. Databases, Etc. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1 (2). Everett, WA: Object Oriented Programming for Smalltalk Application Developers Association.
- Navathe, S., and Cheng, A. 1983. A Methodology for Database Schema Mapping from Extended Entity Relationship Models into the Hierarchical Model, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Ontologic. 1987. *Vbase Technical Overview*. Billerica, MA.
- Oracle. 1989. *Oracle for Macintosh: References, Version 1.1*. Belmont, CA.
- Penny, J., and Stein, J. October 1987. Class Modification in the GemStone Object-Oriented DBMS. *SIGPLAN Notices* vol. 22 (12).
- Peterson, R. 1987. Object-Oriented Database Design, in *Object-Oriented Computing: Implementations* vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Sakai, H. 1983. Entity-Relationship Approach to Logical Database Design, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Skarra, A., and Zdonik, S. November 1986. The Management of Changing Types in an Object-Oriented Database. *SIGPLAN Notices* vol. 21 (11).
- Skarra, A., and Zdonik, S. 1987. Type Evolution in an Object-Oriented Database, in *Research Directions in Object-Oriented Programming* ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Smith, D., and Smith, J. 1980. Conceptual Database Design, in *Tutorial on Software Design Techniques*, 3rd ed. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Smith, J., and Smith, D. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems* vol. 2 (2).
- Smith, K., and Zdonik, S. October 1987. Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems. *SIGPLAN Notices* vol. 22 (12).
- Stein, J. March 1988. Object-Oriented Programming and Database Design. *Dr. Dobbs' Journal of Software Tools for the Professional Programmer*, no. 137.
- Teorey, T., Yang, D., and Fry, J. June 1986. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys* vol. 18 (2).
- Thuraisingham, M. October 1989. Mandatory Security in Object-Oriented Database Systems. *SIGPLAN Notices* vol. 24 (10).

- Veloso, P., and Furtado, A. 1983. View Constructs for the Specification and Design of External Schemas, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Wiebe, D. November 1986. A Distributed Repository for Immutable Persistent Objects. *SIGPLAN Notices* vol. 21 (11).
- Wiederhold, G. December 1986. Views, Objects, and Databases. *IEEE Computer* vol. 19 (12).
- Wile, D., and Allard, D. May 1982. Worlds: an Organizing Structure for Object-bases. *SIGPLAN Notices* vol. 19 (5).
- Zdonik, S., and Maier, D. 1990. *Readings in Object-Oriented Database Systems*. San Mateo, CA: Morgan Kaufmann.
- Zhang, Z., and Mendelson, A. 1983. A Graphical Query Language for Entity-Relationship Databases, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

F. Объектно-ориентированное проектирование

- Abbott, R. November 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26 (11).
- Abbott, R. August 1987. Knowledge Abstraction. *Communications of the ACM* vol. 30 (8).
- Alabios, B. September 1988. Transformation of Data Flow Analysis Models to Object-Oriented Design. *SIGPLAN Notices* vol. 23 (11).
- Beck, K., and Cunningham, W. October 1989. A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices* vol. 24 (10).
- Berard, E. 1986. *An Object-Oriented Design Handbook*. Rockville, MD: EVB Software Engineering.
- Berzins, V., Gray, M., and Naumann, D. May 1986. Abstraction-Based Software Development. *Communications of the ACM* vol. 29 (5).
- Booch, G. September 1981. Describing Software Design in Ada. *SIGPLAN Notices* vol. 16 (9).
- Booch, G. March/April 1982. Object-Oriented Design. *Ada Letters* vol. 1 (3).
- Booch, G. February 1986. Object-Oriented Development. *IEEE Transactions on Software Engineering* vol. 12 (2).
- Booch, G. 1987. *On the Concepts of Object-Oriented Design*. Denver, CO: Rational.
- Booch, G. Summer 1989. What Is and What Isn't Object-Oriented Design. *American Programmer* vol. 2 (7-8).
- Booch, G., Jacobson, I., and Kerth, N. September 1988. Specification and Design Methodologies in Support of Object-Oriented Programming, Course Notes, from *Object-Oriented Programming Systems, Languages, and Applications*. San Diego, CA: OOPSLA'88.
- Boyd, S. July/August 1987. Object-Oriented Design and PAMELA™. *Ada Letters* vol. 7 (4).
- Bruno, G., and Balsamo, A. November 1986. Petri Net-Based Object-Oriented Modelling of Distributed Systems. *SIGPLAN Notices* vol. 21 (11).
- Buhr, R. 1984. *System Design with Ada*. Englewood Cliffs, NJ: Prentice-Hall.

- Buhr, R. 22 August 1988. *Machine Charts for Visual Prototyping in System Design*. SCE Report 88-2. Ottawa, Canada: Carleton University.
- Buhr, R. 14 September 1988. *Visual Prototyping in System Design*. SCE Report 88-14. Ottawa, Canada: Carleton University.
- Buhr, R. 1989. *System Design with Machine Charts: A CAD Approach with Ada Examples*. Englewood Cliffs, NJ: Prentice-Hall.
- Buhr, R., Karam, G., Hayes, C., and Woodside, M. March 1989. Software CAD: A Revolutionary Approach. *IEEE Transactions on Software Engineering* vol. 15 (3).
- Bulman, D. August 1989. An Object-Based Development Model. *Computer Language* vol. 6 (8).
- Cherry, G. 1987. *PAMELA 2: An Ada-Based Object-Oriented Design Method*. Reston, VA: Thought**Tools.
- Cherry, G. 1990. *Software Construction by Object-Oriented Pictures*. Canandaigua, NY: Thought**Tools.
- Clark, R. June 1987. Designing Concurrent Objects. *Ada Letters* vol. 7 (6).
- Comer, E. July 1989. *Ada Box Structure Methodology Handbook*. Melbourne, FL: Software Productivity Solutions.
- Constantine, L. Summer 1989. Object-Oriented and Structured Methods: Towards Integration. *American Programmer* vol. 2 (7-8).
- CRI, CISI Ingenierie, and Matra. 20 June 1987. *HOOD: Hierarchical Object-Oriented Design*. Paris, France.
- Cunningham, W., and Beck, K. November 1986. A Diagram for Object-Oriented Programs. *SIGPLAN Notices* vol. 21 (11).
- Davis, N., Irving, M., and Lee, J. *The Evolution of Object-Oriented Design from Concept to Method*. 1988. Surrey, United Kingdom: Logica Space and Defence Systems Limited.
- Felsing, R. 1987a. *Integrating Object-Oriented Design, Structured Analysis/Structured Design, and Ada for Real-time Systems*. Mt. Pleasant, SC.
- Felsing, R. 1987b. *Object-Oriented Design, Course Notes*. Torrance, CA: Data Processing Management Association.
- Firesmith, D. May 6, 1986. *Object-Oriented Development*. Fort Wayne, Indiana: Magnavox Electronic Systems Co.
- Gane, C. Summer 1989. Object-Oriented Data/Process Modeling. *American Programmer* vol. 2 (7-8).
- Giddings, R. May 1984. Accommodating Uncertainty in Software Design. *Communications of the ACM* vol. 27 (5).
- Gomaa, H. September 1984. A Software Design Method for Real-Time Systems. *Communications of the ACM* vol. 27 (9).
- Gouda, M., Han, Y., Jensen, E., Johnson, W., and Kain, R. November 1977. Towards a Methodology of Distributed Computer System Design. *6th Texas Conference on Computing Systems*. New York, NY: Association of Computing Machinery.
- Grosch, J. December 1983. Type Derivation Graphs – A Way to Visualize the Type Building Possibilities of Programming Languages. *SIGPLAN Notices* vol. 18 (12).
- Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* vol. 8.

- Harel, D. May 1988. On Visual Formalisms. *Communications of the ACM* vol. 31 (5).
- Jackson, M. Summer 1989. Object-Oriented Software. *American Programmerv* vol. 2 (7-8).
- Jacobson, I. August 1985. *Concepts for Modeling Large Real-Time Systems*. Academic dissertation. Stockholm, Sweden: Royal Institute of Technology, Department of Computer Science.
- Jacobson, I. October 1987. Object-Oriented Development in an Industrial Environment. *SIGPLAN Notices* vol. 22 (12).
- Jamsa, K. January 1984. Object-Oriented Design vs. Structured Design – A Student's Perspective. *Software Engineering Notes*, vol. 9 (1).
- Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software, in *Operating Systems*. ed. R. Bayer et. al. New York, NY: Springer-Verlag.
- Kadie, C. 1986. *Refinement Through Classes: A Development Methodology for Object-Oriented Languages*. Urbana, IL: University of Illinois.
- Kaplan, S., and Johnson, R. 21 July 1986. *Designing and Implementing for Reuse*. Urbana, IL: University of Illinois, Department of Computer Science.
- Kay, A. August 1969. *The Reactive Engine*. Salt Lake City, Utah: The University of Utah, Department of Computer Science.
- Kelly, J. 1986 A Comparison of Four Design Methods for Real-Time Systems. *Proceedings of the Ninth International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- Kent, W. 1983. Fact-Based Data Analysis and Design, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Ladden, R. July 1988. A Survey of Issues to Be Considered in the Development of an Object-Oriented Development Methodology for Ada. *Software Engineering Notes* vol. 13 (3).
- Lieberherr, K., and Riel, A. October 1989. Contributions to Teaching Object-Oriented Design and Programming. *SIGPLAN Notices* vol. 24 (10).
- Liskov, B. 1980. A Design Methodology for Reliable Software Systems, in *Tutorial on Software Design Techniques*. 3rd ed. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Mannino, P. April 1987. A Presentation and Comparison of Four Information System Development Methodologies. *Software Engineering Notes*, vol. 12 (2).
- Masiero, P., and Germano, F. July 1988. JSD As an Object-Oriented Design Method. *Software Engineering Notes* vol. 13 (3).
- Meyer, B. March 1987. Reusability: The Case for Object-Oriented Design. *IEEE Software* vol. 4 (2).
- Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
- Meyer, B. 1989. From Structured Programming to Object-Oriented Design: The Road to Eiffel. *Structured Programming* vol. 10 (1).
- Mills, H. June 1988. Stepwise Refinement and Verification in Box-Structured Systems. *IEEE Computer* vol. 21 (6).
- Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis*. Orlando, FL: Academic Press.

- Minkowitz, C., and Henderson, P. March 1987. *Object-Oriented Programming of Discrete Event Simulation Using Petri Nets*. Stirling, Scotland: University of Stirling.
- Mostow, J. Spring 1985. Toward Better Models of the Design Process. *AI Magazine* vol. 6 (1).
- Moulin, B. 1983. The Use of EPAS/IPSO Approach for Integrating Entity Relationship Concepts and Software Engineering Techniques, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Mullin, M. 1989. *Object-Oriented Program Design with Examples in C++*. Reading, MA: Addison-Wesley.
- Nielsen, K. March 1988. *An Object-Oriented Design Methodology for Real-Time Systems in Ada*. San Diego, CA: Hughes Aircraft Company.
- Nielsen, K., and Shumate, K. August 1987. Designing Large Real-Time Systems with Ada. *Communications of the ACM* vol. 30 (8).
- Nies, S. 1986. The Ada Object-Oriented Approach. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Ossher, H. 1987. A Mechanism for Specifying the Structure of Large, Layered, Systems, in *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Parnas, D. 1979. On the Criteria to be Used in Decomposing Systems into Modules. *Classics in Software Engineering*, ed. E. Yourdon. New York, NY: Yourdon Press.
- Parnas, D., Clements, P., and Weiss, D. March 1985. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* vol. SE-11 (3).
- Pasik, A., and Schor, M. January 1984. Object-Centered Representation and Reasoning. *SIGART Newsletter*, no. 87.
- Rajlich, V., and Silva, J. 1987. *Two Object-Oriented Decomposition Methods*. Detroit, Michigan: Wayne State University.
- Ramamoorthy, C., and Sheu, P. Fall 1988. Object-Oriented Systems. *IEEE Expert* vol. 3 (3).
- Reenskaug, T. August 1981. User-Oriented Descriptions of Smalltalk Systems. *Byte* vol. 6 (8).
- Reiss, S. 1987. An Object-Oriented Framework for Conceptual Programming, in *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Richter, C. August 1986. An Assessment of Structured Analysis and Structured Design. *Software Engineering Notes*, vol. 11 (4).
- Rine, D. October 1987. A Common Error in the Object Structure of Object-Oriented Methods. *Software Engineering Notes* vol. 12 (4).
- Ross, R. 1987. *Entity Modeling Techniques and Application*. Boston, MA: Database Research Group.
- Rosson, M., and Gold, E. October 1989. Problem-Solution Mapping in Object-Oriented Design. *SIGPLAN Notices* vol. 24 (10).
- Rumbaugh, J. April 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM* vol. 31 (4).

- Sahraoui, A. 1987. *Towards a Design Approach Methodology Combining OOP and Petri Nets for Software Production*. Toulouse, France: Laboratoire d'Automatique et d'analyses des systemes du C.N.R.S.
- Sakai, H. 1983. A Method for Entity-Relationship Behavior Modeling, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Seidewitz, E. May 1985. *Object Diagrams*. Greenbelt, MD: NASA Goddard Space Flight Center.
- Seidewitz, E., and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Seidewitz, E., and Stark, M. August 1986. *General Object-Oriented Software Development*, Report SEL-86-002. Greenbelt, MD: NASA Goddard Space Flight Center.
- Seidewitz, E., and Stark, M. July/August 1987. Towards a General Object-Oriented Design Methodology. *Ada Letters* vol. 7 (4).
- Seidewitz, E., and Stark, M. 1988. *An Introduction to General Object-Oriented Software Development*. Rockville, MD: Millennium Systems.
- Shilling, J., and Sweeney, P. October 1989. Three Steps to Views: Extending the Object-Oriented Paradigm. *SIGPLAN Notices* vol. 24 (10).
- Shumate, K. 1987. *Layered Virtual Machine/Object-Oriented Design*. San Diego, CA: Hughes Aircraft Company.
- Smith, M., and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects*. Seattle, WA: Boeing Commercial Airplane Support Division.
- Stark, M. April 1986. *Abstraction Analysis: From Structured Analysis to Object-Oriented Design*. Greenbelt, MD: NASA Goddard Space Flight Center.
- Strom, R. October 1986. A Comparison of the Object-Oriented and Process Paradigms. *SIGPLAN Notices* vol. 21 (10).
- Teledyne Brown Engineering. October 1987. *Software Methodology Catalog*, Report MC87-COMM/ADP-0036. Tinton Falls, NJ.
- Thomas, D. May/June 1989. In Search of an Object-Oriented Development Process. *Journal of Object-Oriented Programming* vol. 2 (1).
- Wahl, S. 13 December 1988. Introduction to Object-Oriented Software. *C++ Tutorial Program of the USENIX Conference*. Denver, CO: USENIX Association.
- Wasserman, T., Pircher, P., and Muller, R. December 1988. *An Object-Oriented Structured Design Method for Code Generation*. San Francisco, CA: Interactive Development Environments.
- Wasserman, A., Pircher, P., and Muller, R. Summer 1989. Concepts of Object-Oriented Structured Design. *American Programmer* vol. 2 (7-8).
- Webster, D. December 1988. Mapping the Design Information Representation Terrain. *IEEE Spectrum* vol. 21 (12).
- Williams, L. 1986. *The Object Model in Software Engineering*. Boulder, CO: Software Engineering Research.

- Wirfs-Brock, R., and Wilkerson, B. October 1989. Object-Oriented Design: A Responsibility-Driven Approach. *SIGPLAN Notices* vol. 24 (10).
- Yau, S., and Tsai, J. June 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering* vol. SE-12 (6).
- Zachman, J. 1987. A Framework for Information Systems Architecture. *IBM Systems Journal* vol. 26 (3).
- Zimmerman, R. 1983. Phases, Methods, and Tools – A Triad of System Development, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

Г. Объектно-ориентированное программирование

- Adams, S. July 1986. MetaMethods: The MVC Paradigm. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1 (4). Everett, WA: Object-Oriented Programming for Smalltalk Applications Developers Association.
- Agha, G. October 1986. An Overview of Actor Languages. *SIGPLAN Notices* vol. 21 (10).
- Agha, G. 1988. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press.
- Agha, G., and Hewitt, C. 1987. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming, in *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Aksit, M., and Tripathi, A. September 1988. Data Abstraction Mechanisms in Sina/st. *SIGPLAN Notices* vol. 23 (11).
- Albano, A. June 1983. Type Hierarchies and Semantic Data Models. *SIGPLAN Notices* vol. 18 (6).
- Almes, G., Black, A., Lazowska, E., and Noe, J. January 1985. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering* vol. SE-11 (1).
- Althoff, J. August 1981. Building Data Structures in the Smalltalk-80 System. *Byte* vol. 6 (8).
- Ambler, A. 1980. Gypsy: A Language for Specification and Implementation of Verifiable Programs, in *Programming Language Design*, ed. A. Wasserman. New York, NY: Computer Society Press.
- America, P. 1987. POOL-T: A Parallel Object-Oriented Language, in *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Apple Computer. 1989. *MacApp: The Expandable Macintosh Application*, version 2.0B9. Cupertino, CA.
- *Macintosh Programmer's Workshop Pascal 3.0 Reference*. Cupertino, CA.
- AT&T Bell Laboratories. 1989. *UNIX System V ATT C++ Language System, Release 2.0 Library Manual*. Murray Hill, NJ.
- *UNIX System V ATT C++ Language System, Release 2.0 Product Reference Manual*. Murray Hill, NJ.
- *UNIX System V ATT C++ Language System, Release 2.0 Release Notes*. Murray Hill, NJ.
- *UNIX System V ATT C++ Language System, Release 2.0 Selected Readings*. Murray Hill, NJ.

- Attardi, G. 1987. Concurrent Strategy Execution in Omega, in *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Bach, I. November/December 1982. On the Type Concept of Ada. *Ada Letters* vol. 11 (3).
- Badrinath, B., and Ramamritham, K. May 1988. Synchronizing Transactions on Objects. *IEEE Transactions on Computers* vol. 37 (5).
- Ballard, M., Maier, D., and Wirfs-Brock, A. November 1986. QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods. *SIGPLAN Notices* vol. 21 (11).
- Beaudet, P., and Jenkins, M. June 1988. Simulating the Object-Oriented Paradigm in Nial. *SIGPLAN Notices* vol. 23 (6).
- Bennett, J. October 1987. The Design and Implementation of Distributed Smalltalk. *SIGPLAN Notices* vol. 22 (12).
- Bergin, J., and Greenfield, S. March 1988. What Does Modula-2 Need to Fully Support Object-Oriented Programming? *SIGPLAN Notices* vol. 23 (3).
- Bhaskar, K. October 1983. How Object-Oriented Is Your System? *SIGPLAN Notices* vol. 18 (10).
- Birman, K., Joseph, T., Raeuchle, T., and Abhadi, A. June 1985. Implementing Fault-tolerant Distributed Objects. *IEEE Transactions on Software Engineering* vol. SE-11 (6).
- Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur.
- Black, A., Hutchinson, N., Jul, E., and Levy, H. November 1986. Object Structure in the Emerald System. *SIGPLAN Notices* vol. 21 (11).
- Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, I. July 1986. *Distribution and Abstract Types in Emerald*. Report 86-02-04. Seattle, WA: University of Washington.
- Blaschek, G. 1989. Implementation of Objects in Modula-2. *Structured Programming* vol. 10 (3).
- Blaschek, G., Pomberger, G., and Stritzinger, A. 1989. A Comparison of Object-Oriented Programming Languages. *Structured Programming* vol. 10 (4).
- Block, F., and Chan, N. October 1989. An Extended Frame Language. *SIGPLAN Notices* vol. 24 (10).
- Bobrow, D. November 1984. *If Prolog Is the Answer, What Is the Question?* Palo Alto, California: Xerox Palo Alto Research Center.
- Bobrow, D. 1985. An Overview of KRL, a Knowledge Representation Language, in *Readings in Knowledge Representation*, ed. R. Brachman and H. Levesque. Los Altos, CA: Morgan Kaufmann.
- Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D. September 1988. Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Notices* vol. 23.
- Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. August 1985. *COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming*. Report ISL-85-8. Palo Alto, CA: Xerox Palo Alto Research Center, Intelligent Systems Laboratory.
- Borgida, A. January 1985. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software* vol. 2 (1).

- Borgida, A. October 1986. Exceptions in Object-Oriented Languages. *SIGPLAN Notices* vol. 21 (10).
- Borning, A., and Ingalls, D. 1982a. A Type Declaration and Inference System for Smalltalk. Palo Alto, CA: Xerox Palo Alto Research Center.
- Borning, A., and Ingalls, D. 1982b. Multiple Inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI.
- Bos, J. September 1987. PCOL - A Protocol-Constrained Object Language. *SIGPLAN Notices* vol. 22 (9).
- Briot, J., and Cointe, P. October 1989. Programming with Explicit Metaclasses in Smalltalk. *SIGPLAN Notices* vol. 24 (10).
- Buzzard, G., and Mudge, T. 1987. Object-Based Computing and the Ada Programming Language, in *Object-Oriented Computing: Concepts* vol. 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Canning, P., Cook, W., Hill, W., and Olthoff, W. October 1989. Interfaces for Strongly-Typed Object-Oriented Programming. *SIGPLAN Notices* vol. 24 (10).
- Caudill, P., and Wirfs-Brock, A. November 1986. A Third Generation Smalltalk-80 Implementation. *SIGPLAN Notices* vol. 21 (11).
- Chambers, C., Ungar, D., and Lee, E. October 1989. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *SIGPLAN Notices* vol. 24 (10).
- Clark, K. December 1988. PARLOG and Its Application. *IEEE Transactions on Software Engineering* vol. 14 (12).
- Cleaveland, C. 1980. Programming Languages Considered as Abstract Data Types. *Communications of the ACM*.
- Connor, R., Dearle, A., Morrison, R., and Brown, A. October 1989. An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance. *SIGPLAN Notices* vol. 24 (10).
- Conroy, T., and Pelegri-Llopert, E. 1983. An Assessment of Method-lookup Caches for Smalltalk-80 Implementations, in *Smalltalk-80: Bits of History, Words of Advice*. ed. G. Krasner. Reading, MA: Addison-Wesley.
- Cointe, P. October 1987. Metaclasses Are First Class: the ObjVlisp Model. *SIGPLAN Notices* vol. 22 (12).
- Corradi, A., and Leonardi, L. December 1988. The Role of Opaque Types in Building Abstractions. *SIGPLAN Notices* vol. 23 (12).
- Cox, B. January 1983. The Object-Oriented Pre-compiler. *SIGPLAN Notices* vol. 18 (1).
- Cox, B. October/November 1983. Object-Oriented Programming in C. *Unix Review*.
- Cox, B. January 1984. Message/Object Programming: An Evolutionary Change in Programming Technology. *IEEE Software* vol. 1 (1).
- Cox, B. February/March 1984. Object-Oriented Programming: A Power Tool for Software Craftsmen. *Unix Review*.
- Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley.
- Cox, B., and Hunt, B. August 1986. Objects, Icons, and Software-ICs. *Byte* vol. 11 (8).
- deJong, P. October 1986. Compilation into Actors. *SIGPLAN Notices* vol. 21 (10).

- Deutsch, P. August 1981. Building Control Structures in the Smalltalk-80 System. *Byte* vol. 6 (8).
- Deutsch, P. 1983. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Dewhurst, S., and Stark, K. 1989. *Programming in C++*. Englewood Cliffs, NJ: Prentice Hall.
- Diederich, J., and Milton, J. May 1987. Experimental Prototyping in Smalltalk. *IEEE Software* vol. 4 (3).
- Dixon, R., McKee, T., Schweizer, P., and Vaughn, M. October 1989. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *SIGPLAN Notices* vol. 24 (10).
- Dony, C. August 1988. An Object-Oriented Exception Handling System for an Object-Oriented Language. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Duff, C. August 1986. Designing an Efficient Language. *Byte* vol. 11 (8).
- Dussud, P. October 1989. TICLOS: An Implementation of CLOS for the Explorer Family. *SIGPLAN Notices* vol. 24 (10).
- Eccles, J. 1988. Porting from Common Lisp with Flavors to C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Edelson, D. September 1987. How Objective Mechanisms Facilitate the Development of Large Software Systems in Three Programming Languages. *SIGPLAN Notices* vol. 22 (9).
- Endres, T. May 1985. Clascal - An Object-Oriented Pascal. *Computer Language* vol. 2 (5).
- Filman, R. October 1987. Retrofitting Objects. *SIGPLAN Notices* vol. 22 (12).
- Finzer, W., and Gould, L. June 1984. Programming by Rehearsal. *Byte* vol. 9 (6).
- Foote, B., and Johnson, R. October 1989. Reflective Facilities in Smalltalk-80. *SIGPLAN Notices* vol. 24 (10).
- Freeman-Benson, B. October 1989. A Module Mechanism for Constraints in Smalltalk. *SIGPLAN Notices* vol. 24 (10).
- Fukunaga, K., and Jirose, S. November 1986. An Experience with a Prolog-Based Object-Oriented Language. *SIGPLAN Notices* vol. 21 (11).
- Goldberg, A. August 1981. Introducing the Smalltalk-80 System. *Byte* vol. 6 (8).
- Goldberg, A. September 1988. Programmer as Reader. *IEEE Software* vol. 4 (5).
- Goldberg, A., and Kay, A. March 1976. *Smalltalk-72 Instruction Manual*. Palo Alto, CA: Xerox Palo Alto Research Center.
- Goldberg, A., and Kay, A. 1977. *Methods for Teaching the Programming Language Smalltalk*, Report SSL 77-2. Palo Alto, CA: Xerox Palo Alto Research Center.
- Goldberg, A., and Pope, S. Summer 1989. Object-Oriented Programming Is Not Enough. *American Programmer* vol. 2 (7-8).
- Goldberg, A., and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley.
- Goldberg, A., and Robson, D. 1989. *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley.

- Goldberg, A., and Ross, J. August 1981. Is the Smalltalk-80 System for Children? *Byte* vol. 6 (8).
- Goldstein, T. May 1989. The Object-Oriented Programmer. *The C++ Report* vol. 1 (5).
- Gonsalves, G., and Silvestri, A. December 1986. Programming in Smalltalk-80: Observations and Remarks from the Newly Initiated. *SIGPLAN Notices* vol. 21 (12).
- Gorlen, K. 1989. An Introduction to C++, in *UNIX System V ATT C++ Language System, Release 2.0 Selected Readings*. 1989. Murray Hill, NJ: ATT Bell Laboratories.
- Gougen, J., and Meseguer, J. 1987. Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Graube, N. August 1988. Reflexive Architecture: From ObjVLisp to CLOS. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Hagmann, R. 1983. Preferred Classes: A Proposal for Faster Smalltalk-80 Execution, in *Smalltalk-80: Bits of History, Words of Advice*. ed. G. Krasner. Reading, MA: Addison-Wesley.
- Hailpern, B., and Nguyen, V. 1987. A Model for Object-Based Inheritance, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Halbert, D., and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4 (5).
- Halstead, R. 1987. Object-management on Distributed Systems, in *Object-Oriented Computing: Implementations* vol 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Harland, D., Szyplewski, M., and Wainwright, J. October 1985. An Alternative View of Polymorphism. *SIGPLAN Notices* vol. 20 (10).
- Hendler, J. October 1986. Enhancement for Multiple Inheritance. *SIGPLAN Notices* vol. 21 (10).
- Hines, T., and Unger, E. 1986. *Conceptual Object-Oriented Programming*. Manhattan, Kansas: Kansas State University.
- Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, New York, NY: Association of Computing Machinery.
- Ingalls, D. August 1981a. Design Principles Behind Smalltalk. *Byte* vol. 6 (8).
- Ingalls, D. August 1981b. The Smalltalk Graphics Kernel. *Byte* vol. 6 (8).
- Ingalls, D. 1983. The Evolution of the Smalltalk Virtual Machine, in *Smalltalk-80: Bits of History, Words of Advice*. ed. G. Krasner. Reading, MA: Addison-Wesley.
- Ingalls, D. November 1986. A Simple Technique for Handling Multiple Polymorphism. *SIGPLAN Notices* vol. 21 (11).
- Ishikawa, Y., and Tokoro, M. 1987. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Jackson, M. May 1988. Objects and Other Subjects. *SIGPLAN Notices* vol. 23 (5).

- Jacky, J., and Kalet, I. September 1987. An Object-Oriented Programming Discipline for Standard Pascal. *Communications of the ACM* vol. 30 (9).
- Jacobson, I. November 1986. Language Support for Changeable, Large, Real-Time Systems. *SIGPLAN Notices* vol. 21 (11).
- Jeffery, D. February 1989. Object-Oriented Programming in ANSI C. *Computer Language*.
- Jenkins, M., and Glasgow, J. January 1986. Programming Styles in Nial. *IEEE Software* vol. 3 (1).
- Johnson, R. November 1986. Type-Checking Smalltalk. *SIGPLAN Notices* vol. 21 (11).
- Johnson, R., Graver, J., and Zurawski, L. September 1988. TS: An Optimizing Compiler for Smalltalk. *SIGPLAN Notices* vol. 23 (11).
- Kaehler, T. November 1986. Virtual Memory on a Narrow Machine for an Object-Oriented Language. *SIGPLAN Notices* vol. 21 (11).
- Kaehler, T., and Patterson, D. 1986. *A Taste of Smalltalk*. New York, NY: W. W. Norton.
- Kaehler, T., and Patterson, D. August 1986. A Small Taste of Smalltalk. *Byte* vol. 11 (8).
- Kahn, K., Tribble, E., Miller, M., and Bobrow, D. November 1986. Objects in Concurrent Logic Programming Languages. *SIGPLAN Notices* vol. 21 (11).
- Kahn, K., Tribble, E., Miller, M., and Bobrow, D. 1987. Vulcan: Logical Concurrent Objects, in *Research Directions in Object-Oriented Programming* ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Kaiser, G., and Garlan, D. October 1987. MELDing Data Flow and Object-Oriented Programming. *SIGPLAN Notices* vol. 22 (12).
- Kalme, C. 27 March 1986. *Object-Oriented Programming: A Rule-Based Perspective*. Los Angeles, CA: Inference Corporation.
- Kay, A. *New Directions for Novice Programming in the 1980s*. Palo Alto, CA: Xerox Palo Alto Research Center.
- Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley.
- Kelly, K., Rischer, R., Pleasant, M., Steiner, D., McGrew, C., Rowe, J., and Rubin, M. 30 March 1986. *Textual Representations of Object-Oriented Programs for Future Programmers*. Palo Alto, CA: Xerox AI Systems.
- Kempf, J., Harris, W., D'Souza, R., and Snyder, A. October 1987. Experience with CommonLoops. *SIGPLAN Notices* vol. 22 (12).
- Kempf, R. October 1987. Teaching Object-Oriented Programming with the KEE System. *SIGPLAN Notices* vol. 22 (12).
- Khosafian, S., and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21 (11).
- Kilian, M. April 1987. *An Overview of the Trellis/Owl Compiler*. Hudson, MA: Digital Equipment Corporation.
- Kimminau, D., and Seagren, M. 1987. *Comparison of Two Prototype Developments Using Object-Based Programming*. Naperville, IL: AT&T Bell Laboratories.
- Koshmann, T., and Evens, M. July 1988. Bridging the Gap Between Object-Oriented and Logic Programming. *IEEE Software* vol. 5 (4).
- Koskimies, K., and Paakki, J. July 1987. TOOLS: A Unifying Approach to Object-Oriented Language Interpretation. *SIGPLAN Notices* vol. 22 (7).

- Knowledge Systems Corporation. 1987. *PluggableGauges Version 1.0 User Manual*. Cary, NC.
- Knudsen, J. August 1988. Name Collision in Multiple Classification Hierarchies. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Knudsen, J., and Madsen, O. August 1988. Teaching Object-Oriented Programming Is More than Teaching Object-Oriented Programming Languages. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Krasner, G. August 1981. The Smalltalk-80 Virtual Machine. *Byte* vol. 6 (8).
- Krasner, G., ed. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley.
- Krasner, G., and Pope, S. August/September 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* vol. 1 (3).
- Kristensen, B., Madsen, O., Moller-Pedersen, B., and Nygaard, K. 1987. The BETA Programming Language, in *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner. Cambridge, MA: The MIT Press.
- LaLonde, W. April 1989. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems* vol. 11 (2).
- LaLonde, W., Thomas, D., and Pugh, J. November 1986. An Exemplar Based Smalltalk. *SIGPLAN Notices* vol. 21 (11).
- Lang, K., and Peralmutter, B. November 1986. Oaklisp: an Object-Oriented Scheme with First Class Types. *SIGPLAN Notices* vol. 21 (11).
- Laursen, J., and Atkinson, R. October 1987. Opus: A Smalltalk Production System. *SIGPLAN Notices* vol. 22 (12).
- Lieberherr, K., and Holland, I. March 1989. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices* vol. 24 (3).
- Lieberherr, K., and Holland, I. September 1989. Assuring Good Style for Object-Oriented Programs. *IEEE Software* vol. 6 (5).
- Lieberherr, K., Holland, I., Lee, G., and Riel, A. June 1988. An Objective Sense of Style. *IEEE Computer* vol. 21 (6).
- Lieberman, H. November 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *SIGPLAN Notices* vol. 21 (11).
- Lieberman, H. 1987. Concurrent Object-Oriented Programming in Act 1, in *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Lieberman, H., Stein, L., and Ungar, D. May 1988. Of Types and Prototypes: The Treaty of Orlando. *SIGPLAN Notices* vol. 23 (5).
- Lim, J., and Johnson, R. April 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices* vol. 24 (4).
- Linowes, J. August 1988. It's an Attitude. *Byte* vol. 13 (8).
- Lippman, S. 1989. *C++ Primer*. Reading, MA: Addison-Wesley.
- Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, R., and Snyder, R. 1981. *CLU Reference Manual*. New York, NY: Springer-Verlag.

- Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1980. Abstraction Mechanisms in CLU, in *Programming Language Design*. ed. A. Wasserman. New York, NY: Computer Society Press.
- Lujun, S., and Zhongxiu. August 1987. An Object-Oriented Programming Language for Developing Distributed Software. *SIGPLAN Notices* vol. 22 (8).
- MacLennan, B. 1987. Values and Objects in Programming Languages., in *Object-Oriented Computing: Concepts* vol. 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Madsen, O. 1987. Block Structure and Object-Oriented Languages, in *Research Directions in Object-Oriented Programming*. ed. B. Shriver and P. Wegner. Cambridge, MA: The MIT Press.
- Madsen, O., and Moller-Pedersen, B. August 1988. What Object-Oriented Programming May Be – And What It Does Not Have To Be. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Madsen, O., and Moller-Pedersen, B. October 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. *SIGPLAN Notices* vol. 24 (10).
- Marcus, R. November 1985. Generalized Inheritance. *SIGPLAN Notices* vol. 20 (11).
- Markowitz, V., and Raz, Y. 1983. Eroll: An Entity-Relationship, Role-Oriented Query Language, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Mellender, F. October 1988. An Integration of Logic and Object-Oriented Programming. *SIGPLAN Notices* vol. 23 (10).
- Methfessel, R. April 1987. Implementing an Access and Object-Oriented Paradigm in a Language That Supports Neither. *SIGPLAN Notices* vol. 22 (4).
- Meyer, B. November 1986. Genericity versus Inheritance. *SIGPLAN Notices* vol. 21 (11).
- Meyer, B. February 1987. Eiffel: Programming for Reusability and Extendability. *SIGPLAN Notices* vol. 22 (2).
- Meyer, B. November/December 1988. Harnessing Multiple Inheritance. *Journal of Object-Oriented Programming* vol. 1 (4).
- Micallef, J. April/May 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1 (1).
- Minsky, N., and Rozenshtein, D. October 1987. A Law-Based Approach to Object-Oriented Programming. *SIGPLAN Notices* vol. 22 (12).
- Minsky, N., and Rozenshtein, D., October 1989. Controllable Delegation: An Exercise in Law-Governed Systems. *SIGPLAN Notices* vol. 24 (10).
- Miranda, E. October 1987. BrouHaHa – A Portable Smalltalk Interpreter. *SIGPLAN Notices* vol. 22 (12).
- Mittal, S., Bobrow, D., and Kahn, K. November 1986. Virtual Copies: At the Boundary Between Classes and Instances. *SIGPLAN Notices* vol. 21 (11).
- Moon, D. November 1986. Object-Oriented Programming with Flavors. *SIGPLAN Notices* vol. 21 (11).
- Mossenbock, H., and Templ, J. 1989. Object Oberon – A Modest Object-Oriented Language. *Structured Programming* vol. 10 (4).

- Mudge, T. March 1985. Object-Based Computing and the Ada Language. *IEEE Computer* vol. 18 (3).
- Nierstrasz, O. October 1987. Active Objects in Hybrid. *SIGPLAN Notices* vol. 22 (12).
- Novak, G. June 1983. Data Abstraction in GLISP. *SIGPLAN Notices* vol. 18 (6).
- Novak, G. Fall 1983. GLISP: A Lisp-Based Programming System with Data Abstraction. *AI Magazine* vol. 4 (3).
- Nygaard, K., and Dahl, O-J. 1981. The Development of the Simula Languages, in *History of Programming Languages*. ed. R. Wexelblat. New York, NY: Academic Press.
- Nygaard, K. October 1986. Basic Concepts in Object-Oriented Programming. *SIGPLAN Notices* vol. 21 (10).
- Object-Oriented Programming Workshop. October 1986. *SIGPLAN Notices* vol. 21 (10).
- O'Brien, P. 15 November 1985. *Trellis Object-Based Environment: Language Tutorial*. Hudson, MA: Digital Equipment Corporation.
- Olthoff, W. 1986. *Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience*. Kaiserslautern, West Germany: University of Kaiserslautern.
- Osterbye, K. June/July 1988. Active Objects: An Access-Oriented Framework for Object-Oriented Languages. *Journal of Object-Oriented Programming* vol. 1 (2).
- Paepcke, A. October 1989. PCLOS: A Critical Review. *SIGPLAN Notices* vol. 24 (10).
- Parc Place Systems. 1988. *The Smalltalk-80 Programming System Version VI 2.3*. Palo Alto, CA.
- Pascoe, G. August 1986. Elements of Object-Oriented Programming. *Byte* vol. 11 (8).
- Pascoe, G. November 1986. Encapsulators: A New Software Paradigm in Smalltalk-80. *SIGPLAN Notices* vol. 21 (11).
- Perez, E. September/October 1988. Simulating Inheritance with Ada. *Ada Letters* vol. 8 (7).
- Peterson, G. ed. 1987. *Object-Oriented Computing Concepts*. New York, NY: Computer Society Press of the IEEE.
- Pinson, L., and Wiener, R. 1988. *An Introduction to Object-Oriented Programming and Smalltalk*. Reading, MA: Addison-Wesley.
- Pohl, I. 1989. *C++ for C Programmers*. Redwood City, CA: Benjamin/Cummings.
- Pokkunuri, B. November 1989. Object-Oriented Programming. *SIGPLAN Notices* vol. 24 (11).
- Pountain, D. August 1986. Object-Oriented FORTH. *Byte* vol. 11 (8).
- Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. August 1988. New York, NY: Springer-Verlag.
- Proceedings of OOPSLA'86: Object-Oriented Programming Systems, Languages, and Applications*. November 1986. *SIGPLAN Notices* vol. 21 (11).
- Proceedings of OOPSLA'87: Object-Oriented Programming Systems, Languages, and Applications*. October 1987. *SIGPLAN Notices* vol. 22 (12).
- Proceedings of OOPSLA'88: Object-Oriented Programming Systems, Languages, and Applications*. September 1988. *SIGPLAN Notices* vol. 23 (11).
- Proceedings of OOPSLA'89: Object-Oriented Programming Systems, Languages, and Applications*. October 1989. *SIGPLAN Notices* vol. 24 (10).

- Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*. April 1989. *SIGPLAN Notices* vol. 24 (4).
- Proceedings of the USENIX Association C++ Workshop*. November 1987. Berkeley, CA: USENIX Association.
- Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling*. 1980. *SIGPLAN Notices* vol. 16 (1).
- Pugh, J. March 1984. Actors – The Stage is Set. *SIGPLAN Notices* vol. 19 (3).
- Rathke, C. 1986. *ObjTalk: Repräsentation von Wissen in einer objektorientierten Sprache*. Stuttgart, West Germany: Institut für Informatik der Universität Stuttgart.
- Rentsch, T. September 1982. Object-Oriented Programming. *SIGPLAN Notices* vol. 17 (12).
- Rettig, M., Morgan, T., Jacobs, J., and Wimberly, D. January 1989. Object-Oriented Programming in AI. *AI Expert*.
- Robson, D. August 1981. Object-Oriented Software Systems. *Byte* vol. 6 (8).
- Rumbaugh, J. October 1987. Relations as Semantic Constructs in an Object-Oriented Language. *SIGPLAN Notices* vol. 22 (12).
- Russo, V., and Kaplan, S. 1988. A C++ Interpreter for Scheme. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Sakkinen, M. August 1988. On the Darker Side of C++. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Sakkinen, M. December 1988. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices* vol. 23 (12).
- Saltzer, J. 1979. Naming and Binding of Objects, in *Operating Systems*. ed. R. Bayer et. al. New York, NY: Springer-Verlag.
- Sandberg, D. November 1986. An Alternative To Subclassing. *SIGPLAN Notices* vol. 21 (11).
- Sandberg, D. October 1988. Smalltalk and Exploratory Programming. *SIGPLAN Notices* vol. 23 (10).
- Saunders, J. March/April 1989. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1 (6).
- Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. November 1986. An Introduction to Trellis/Owl. *SIGPLAN Notices* vol. 21 (11).
- Schaffert, C., Cooper, T., and Wilpolt, C. November 25, 1985. *Trellis Object-Based Environment: Language Reference Manual*. Hudson, MA: Digital Equipment Corporation.
- Schmucker, K. 1986a. MacApp: An Application Framework. *Byte* vol. 11 (8).
- Schmucker, K. 1986b. Object-Oriented Languages for the Macintosh. *Byte* vol. 11 (8).
- Schmucker, K. 1986c. *Object-Oriented Programming for the Macintosh*. Hashbrouk Heights, NJ: Hayden.
- Schriver, B., and Wegner, P. eds. 1987. *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press.
- Seidewitz, E. October 1987. Object-Oriented Programming in Smalltalk and Ada. *SIGPLAN Notices* vol. 22 (12).

- Shafer, D. 1988. *HyperTalk Programming*. Indianapolis, IN: Hayden Book.
- Shah, A., Rumbaugh, J., Hamel, J., and Borsari, R. October 1989. DSM: An Object-Relationship Modeling Language. *SIGPLAN Notices* vol. 24 (10).
- Shammas, N. October 1988. Smalltalk a la C. *Byte* vol. 13 (10).
- Shan, Y. October 1989. An Event-Driven Model-View-Controller Framework for Smalltalk. *SIGPLAN Notices* vol. 24 (10).
- Shaw, M. 1981. *ALPHARD: Form and Content*. New York, NY: Springer-Verlag.
- Shibayama, E. September 1988. How to Invent Distributed Implementation Schemes of an Object-Based Concurrent Language - A Transformational Approach. *SIGPLAN Notices* vol. 23 (11).
- Shibayama, E., and Yonezawa, A. 1987. Distributed Computing in ABCL/1, in *Object-Oriented Concurrent Programming*, ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Shopiro, J. 13 December 1988. Programming Techniques with C++. *C++ Tutorial Program of the USENIX Conference*. Denver, CO: USENIX Association.
- Simonian, R., and Crone, M. November/December 1988. InnovAda: True Object-Oriented Programming in Ada. *Journal of Object-Oriented Programming* vol. 1 (4).
- Software Productivity Solutions. 1988. *Classical-Ada User Manual*. Melbourne, FL.
- Snyder, A. February 1985. *Object-Oriented Programming for Common Lisp*. Report ATC-85-1. Palo Alto, CA: Hewlett-Packard.
- Snyder, A. November 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. *SIGPLAN Notices* vol. 21 (11).
- Snyder, A. 1987. Inheritance and the Development of Encapsulated Software Components, in *Research Directions in Object-Oriented Programming*, ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Stankovic, J. April 1982. Software Communication Mechanisms: Procedure Calls Versus Messages. *IEEE Computer* vol. 15 (4).
- Stefik, M., and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations. *AI Magazine* vol. 6 (4).
- Stefik, M., Bobrow, D., Mittal, S., and Conway, L. Fall 1983. Knowledge Programming in Loops. *AI Magazine* vol. 4 (3).
- Stein, L. October 1987. Delegation Is Inheritance. *SIGPLAN Notices* vol. 22 (12).
- Stroustrup, B. January 1982. Classes: An Abstract Data Type Facility for the C Language. *SIGPLAN Notices* vol. 17 (1).
- Stroustrup, B. October 1986. An Overview of C++. *SIGPLAN Notices* vol. 21 (10).
- Stroustrup, B. 1986. *The C++ Programming Language*. Reading, MA: Addison-Wesley.
- Stroustrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM: USENIX Association.
- Stroustrup, B. November 1987. The Evolution of C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM: USENIX Association.
- Stroustrup, B. May 1988. What Is Object-Oriented Programming? *IEEE Software* vol. 5 (3).
- Stroustrup, B. August 1988. A Better C? *Byte* vol. 13 (8).
- Stroustrup, B. 1988. Parameterized Types for C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

- Suzuki, N. 1981. Inferring Types in Smalltalk, *Proceedings of the 8th Annual Symposium of ACM Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Suzuki, N., and Terada, M. 1983. Creating Efficient Systems for Object-Oriented Languages. *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Symposium on Actor Languages. October 1980. *Creative Computing*.
- Tektronix. 1988. *Modular Smalltalk*.
- Tesler, L. August 1986. Programming Experiences. *Byte* vol. 11 (8).
- The Smalltalk-80 System. August 1981. *Byte* vol. 6 (8).
- Thomas, D. March 1989. What's in an Object? *Byte* vol. 14 (3).
- Tieman, M. 1 May 1988. *User's Guide to GNU C++*. Cambridge, MA: Free Software Foundation.
- Tokoro, M., and Ishikawa, Y. October 1986. Concurrent Programming in Orient84/K: An Object-Oriented Knowledge Representation Language. *SIGPLAN Notices* vol. 21 (10).
- Touati, H. May 1987. Is Ada an Object-Oriented Programming Language? *SIGPLAN Notices* vol. 22 (5).
- Tripathi, A., and Berge, E. An Implementation of the Object-Oriented Concurrent Programming Language SINA. *Software - Practice and Experience* vol. 19 (3).
- Ungar, D. September 1988. Are Classes Obsolete? *SIGPLAN Notices* vol. 23 (11).
- Ungar, D., and Smith, R. October 1987. Self: The Power of Simplicity. *SIGPLAN Notices* vol. 22 (12).
- U. S. Department of Defense. February 1983. *Reference Manual for the Ada Programming Language*. Washington, D.C.: Ada Joint Program Office.
- van den Bos, J., and Laffra, C. October 1989. PROCOL: A Parallel Object Language with Protocols. *SIGPLAN Notices* vol. 24 (10).
- Vaucher, J., Lapalme, G., and Malenfant, J. August 1988. SCOOP: Structured Concurrent Object-Oriented Prolog. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Warren, S., and Abbe, D. May 1980. Presenting Rosetta Smalltalk. *Datamation*.
- Watanabe, T., and Yonezawa, A. September 1988. Reflection in an Object-Oriented Concurrent Language. *SIGPLAN Notices* vol 23 (11).
- Wegner, P. October 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices* vol. 22 (12).
- Wegner, P. January 1988. Workshop on Object-Oriented Programming at ECOOP 1987. *SIGPLAN Notices* vol. 23 (1).
- Wiener, R. June 1987. Object-Oriented Programming in C++ - A Case Study. *SIGPLAN Notices* vol. 22 (6).
- Williams, G. Summer 1989. Designing the Future: The Power of Object-Oriented Programming. *American Programmer* vol. 2 (7-8).
- Wilson, R. 1 November 1987. Object-Oriented Languages Reorient Programming Techniques. *Computer Design* vol. 26 (20).
- Winston, P., and Horn, B. 1989. *Lisp*. 3rd ed. Reading, MA: Addison-Wesley.

- Wirfs-Brock, R. and Wilkerson, B. September 1988. An Overview of Modular Smalltalk. *SIGPLAN Notices* vol. 23 (11).
- Wirth, N. June 1987. Extensions of Record Types. *SIGCSE Bulletin* vol. 19 (2).
- Wirth, N. July 1988a. From Modula to Oberon. *Software - Practice and Experience* vol. 18 (7).
- Wirth, N. July 1988b. The Programming Language Oberon. *Software - Practice and Experience* vol. 18 (7).
- Wolf, W. September 1989. A Practical Comparison of Two Object-Oriented Languages. *IEEE Software* vol. 6 (5).
- Yokote, Y., and Tokoro, M. November 1986. The Design and Implementation of Concurrent Smalltalk. *SIGPLAN Notices* vol. 21 (11).
- Yokote, Y., and Tokoro, M. October 1987. Experience and Evolution of Concurrent Smalltalk. *SIGPLAN Notices* vol. 22 (12).
- Yonezawa, A., and Tokoro, M. eds. 1987. *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press.
- Yonezawa, A., Briot, J., and Shibayama, E. November 1986. Object-Oriented Concurrent Programming in ABCL/1. *SIGPLAN Notices* vol. 21 (11).
- Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. 1987. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Yourdon, E. February 1990. Object-Oriented COBOL. *American Programmer* vol. 3 (2).
- Zave, P. September 1989. A Compositional Approach to Multiparadigm Programming. *IEEE Software* vol. 6 (5).

Н. Разработка программных продуктов

- Abelson, H., and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press.
- Appleton, D. 15 January 1986. Very Large Projects. *Datamation*.
- Aron, J. 1974a. *The Program Development Process: The Individual Programmer*. Vol. 1. Reading, MA: Addison-Wesley.
- Aron, J. 1974b. *The Program Development Process: The Programming Team*. Vol. 2. Reading, MA: Addison-Wesley.
- Ben-Ari, M. 1982. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice-Hall.
- Boehm, B. August 1986. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, vol. 11 (4).
- Boehm-Davis, D., and Ross, L. October 1984. *Approaches to Structuring the Software Development Process*, Report GEC/DIS/TR-84-B1V-1. Arlington, VA: General Electric.
- Booch, G. 1986. *Software Engineering with Ada*. Menlo Park, CA: Benjamin/Cummings.
- Brooks, F. 1975. *The Mythical Man-Month*. Reading, MA: Addison-Wesley.
- Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* vol. 20 (4).

- Curtis, B. 17 May 1989. . . . *But You Have To Understand, This Isn't The Way We Develop Software At Our Company*. MCC Technical Report Number STP-203-89. Austin, TX: Microelectronics and Computer Technology Corporation.
- Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press.
- Davis, A., Bersoff, E., and Comer, E. October 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering* vol. 14 (10).
- Davis, C., Jajodia, S., Ng, P., and Yeh, R. eds. 1983. *Entity-Relationship Approach to Software Engineering*. Amsterdam, The Netherlands: Elsevier Science.
- DeMarco, T., and Lister, T. 1987. *Peopleware*. New York, NY: Dorset House.
- DeRemer, F., and Kron, H. 1980. Programming-in-the-Large versus Programming-in-the-Small. *Tutorial on Software Design Techniques* 3rd ed. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Dijkstra, E. 1979. Programming Considered as a Human Activity. in *Classics in Software Engineering* ed. E. Yourdon. New York, NY: Yourdon Press.
- Dijkstra, E. 1982. *Selected Writings on Computing: A Personal Perspective*. New York, NY: Springer-Verlag.
- Dowson, M. August 1986. The Structure of the Software Process. *Software Engineering Notes*, vol. 11 (4).
- Dreger, B. 1989. *Function Point Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- Eastman, N. 1984. Software Engineering and Technology. *Technical Directions* vol. 10 (1). Bethesda, MD: IBM Federal Systems Division.
- Foster, C. 1981. *Real-Time Programming*. Reading, MA: Addison-Wesley.
- Freeman, P. 1975. *Software Systems Principles*. Chicago, IL: Science Research Associates.
- Freeman, P., and Wasserman, A. eds. 1983. *Tutorial on Software Design Techniques* 4th ed. New York, NY: Computer Society Press of the IEEE.
- Glass, R. 1982. *Modern Programming Practices: A Report from Industry*. Englewood Cliffs, NJ: Prentice-Hall.
- Glass, R. 1983. *Real-Time Software*. Englewood Cliffs, NJ: Prentice-Hall.
- Hansen, P. 1977. *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall.
- Hoare, C. April 1984. Programming: Sorcery or Science? *IEEE Software* vol. 1 (2).
- Holt, R., Lazowska, E., Graham, G., and Scott, M. 1978. *Structured Concurrent Programming*. Reading, MA: Addison-Wesley.
- Humphrey, W. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.
- Jackson, M. 1975. *Principles of Program Design*. Orlando, FL: Academic Press.
- Jackson, M. 1983. *System Development*. Englewood Cliffs, NJ: Prentice-Hall.
- Jensen, R., and Tonies, C. 1979. *Software Engineering*. Englewood Cliffs, NJ: Prentice-Hall.
- Jones, C. September 1984. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering* vol. SE-10 (5).

- Kishida, K., Teramoto, M., Torri, K., and Urano, Y. September 1988. Quality Assurance Technology in Japan. *IEEE Software* vol. 4 (5).
- Lammers, S. 1986. *Programmers at Work*. Redmond, WA: Microsoft Press.
- Ledgard, H. Summer 1985. Programmers: The Amateur vs. the Professional. *Abacus* vol. (4).
- Linger, R., and Mills, H. 1977. On the Development of Large Reliable Programs, in *Current Trends in Programming Methodology: Software Specification and Design* vol. 1. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Linger, R., Mills, H., and Witt, B. 1979. *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley.
- Liskov, B., and Guttig, J. 1986. *Abstraction and Specification in Program Development*. Cambridge, MA: The MIT Press.
- Lorin, H. 1972. *Parallelism in Hardware and Software*. Englewood Cliffs, NJ: Prentice-Hall.
- Martin, J., and McClure, C. 1988. *Structured Techniques: The Basis for CASE*. Englewood Cliffs, NJ: Prentice-Hall.
- Mascot, Version 3.1, The Official Handbook of*. June 1987. London, England: Crown Copyright.
- Mellichamp, D. 1983. *Real-Time Computing*. New York, NY: Van Nostrand Reinhold.
- Mills, H. November 1986. Structured Programming: Retrospect and Prospect. *IEEE Software* vol. 3 (6).
- Munck, R. 1985. Toward Large Software Systems That Work. *Proceedings of the AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference*. Menlo Park, CA: AIAA.
- Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold.
- Newport, J. 28 April 1986. A Growing Gap in Software. *Fortune*.
- Office of the Under Secretary of Defense for Acquisition. September 1987. *Report of the Defense Science Board Task Force on Military Software*. Washington, D.C.
- Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.
- Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.
- Parnas, D. July 1985a. Why Conventional Software Development Does Not Produce Reliable Programs. *Software Aspects of Strategic Defense Systems*, Report DCS-47-IR. Victoria, Canada: University of Victoria.
- Parnas, D. July 1985b. Why Software is Unreliable. *Software Aspects of Strategic Defense Systems*, Report DCS-47-IR. Victoria, Canada: University of Victoria.
- Parnas, D. December 1985. Software Aspects of Strategic Defense Systems. *Communications of the ACM* vol. 28 (12).
- Peters, L. 1981. *Software Design*. New York, NY: Yourdon Press.
- Pressman, R. 1987. *Software Engineering: A Practitioner's Approach*. 2nd ed. New York, NY: McGraw-Hill.
- Ramamoorthy, C., Garg, V., and Prakask, A. July 1986. Programming in the Large. *IEEE Transactions on Software Engineering* vol. SE-12 (7).

- Ross, D., Goudenough, J., and Irvine, C. 1980. *Software Engineering: Process, Principles, and Goals. Tutorial on Software Design Techniques*. 3rd ed. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Sommerville, I. 1989. *Software Engineering*. 3rd ed. Wokingham, England: Addison- Wesley.
- Spector, A., and Gifford, D. April 1986. Computer Science Perspective of Bridge Design. *Communications of the ACM* vol. 29 (4).
- Stevens, W. Myers, G., and Constantine, L. 1979. Structured Design, in *Classics in Software Engineering*. ed. E. Yourdon. New York, NY: Yourdon Press.
- The Software Trap: Automate – Or Else. 9 May 1988. *Business Week*.
- U. S. Department of Defense. 30 July 1982. *Report of the DoD Joint Service Task Force on Software Problems*. Washington, D.C.
- Vick, C., and Ramamoorthy, C. 1984. *Software Engineering*. New York, NY: Van Nostrand Reinhold.
- Vonk, R. 1990. *Prototyping*. Englewood Cliffs, NJ: Prentice-Hall.
- Ward, P., and Mellor, S. 1985. *Structured Development for Real-Time Systems: Introduction and Tools*. Englewood Cliffs, NJ: Yourdon Press.
- Wegner, P. 1980. *Research Directions in Software Technology*. Cambridge, MA: The MIT Press.
- Wegner, P. July 1984. Capital-intensive Software Technology. *IEEE Software* vol. 1 (3).
- Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall.
- Yeh, R. ed. 1977. *Current Trends in Programming Methodology: Software Specification and Design*. Englewood Cliffs, NJ: Prentice-Hall.
- Yourdon, E. 1975. *Techniques of Program Structure and Design*. Englewood Cliffs, NJ: Prentice-Hall.
- Yourdon, E. 1979. ed. *Classics in Software Engineering*. New York, NY: Yourdon Press.
- Yourdon, E. 1989a. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- Yourdon, E. 1989b. *Structured Walkthroughs*. Englewood Cliffs, NJ: Prentice-Hall.
- Yourdon, E. August 1989. The Year of the Object. *Computer Language* vol. 6 (8).
- Yourdon, E. Summer 1989. Object-Oriented Observations. *American Programmer* vol. 2 (7-8).
- Yourdon, E., and Constantine, L. 1979. *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall.
- Zave, P. February 1984. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM* vol. 27 (2).
- Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10 (2).

I. Специальная информация

- Rand, Ayn. 1979. *Introduction to Objectivist Epistemology*. New York, NY: New American Library.
- Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. 2nd ed. Ann Arbor, MI: The General Systemantics Press.

- Gleick, J. 1987. *Chaos*. New York, NY: Penguin Books.
- Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group.
- Hofstadter, D. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY: Vintage Books.
- Inside Macintosh* Volumes 1-5. 1988. Reading, MA: Addison-Wesley.
- Meyer, C., and Matyas. 1982. *Cryptography*. New York, NY: John Wiley and Sons.
- Peter, L. 1986. *The Peter Pyramid*. New York, NY: William Morrow.
- Petroski, H. 1985. *To Engineer Is Human*. New York, NY: St. Martin's Press.
- Sears, F., Zemansky, M., and Young, H. 1987. *University Physics*. 7th ed. Reading, MA: Addison-Wesley.
- Wagner, J. 1986. *The Search for Signs of Intelligent Life in the Universe*. New York, NY: Harper and Row.
- Whitehead, A. 1958. *An Introduction to Mathematics*. New York, NY: Oxford University Press.

Ж. Теория

- Aho, A., Hopcroft, J., and Ullman, J. 1974. *The Design and Analysis of Computer Programs*. Reading, MA: Addison-Wesley.
- Almarode, J. October 1989. Rule-Based Delegation for Prototypes. *SIGPLAN Notices* vol. 24 (10).
- Appelbe, W., and Ravn, A. April 1984. Encapsulation Constructs in Systems Programming Languages. *ACM Transactions on Programming Languages and Systems* vol. 6 (2).
- Averill, E. April 1982. Theory of Design and Its Relationship to Capacity Measurement. *Proceedings of the Fourth Annual International Conference on Computer Capacity Management*. San Francisco, CA: Association of Computing Machinery.
- Barr, A., and Feigenbaum, E. 1981. *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufmann.
- Bastani, F., and Iyengar, S. March 1987. The Effect of Data Structures on the Logical Complexity of Programs. *Communications of the ACM* vol. 30 (3).
- Bastani, F., Hilal, W., and Sitharama, S. October 1987. Efficient Abstract Data Type Components for Distributed and Parallel Systems. *IEEE Computer* vol. 20 (10).
- Belkhouche, B., and Urban, J. May 1986. Direct Implementation of Abstract Data Types from Abstract Specifications. *IEEE Transactions on Software Engineering* vol. SE-12 (5).
- Bensley, E., Brando, T., and Prelle, M. September 1988. An Execution Model for Distributed Object-Oriented Computation. *SIGPLAN Notices* vol. 23 (11).
- Bertziss, A. 1980. Data Abstraction, Controlled Iteration, and Communicating Processes. *Communications of the ACM*.
- Bishop, J. 1986. *Data Abstraction in Programming Languages*. Wokingham, England: Addison-Wesley.
- Boehm, H., Demers, A., and Donahue, J. October 1980. *An Informal Description of Russell*. Technical Report TR 80-430. Ithaca, NY: Cornell University

- Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. October 1987. Constraint Hierarchies. *SIGPLAN Notices* vol. 22 (12).
- Boute, R. January 1988. Systems Semantics: Principles, Applications, and Implementation. *ACM Transactions on Programming Languages and Systems* vol. 10. (1).
- Brachman, R. October 1983. What Is-a Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer* vol. 16 (10).
- Brachman, R., and Levesque, H. eds. 1985. *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufmann.
- Bruce, K., and Wegner, P. October 1986. An Algebraic Model of Subtypes in Object-Oriented Languages. *SIGPLAN Notices* vol. 21 (10).
- Cardelli, L., and Wegner, P. December 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* vol. 17 (4).
- Claybrook, B., and Wyckof, M. 1980. Module: an Encapsulation Mechanism for Specifying and Implementing Abstract Data Types. *Communications of the ACM*.
- Cline, A., and Rich, E. December 1983. *Building and Evaluating Abstract Data Types*, Report TR-83-26. Austin, TX: University of Texas, Department of Computer Sciences.
- Cohen, A. January 1984. Data Abstraction, Data Encapsulation, and Object-Oriented Programming. *SIGPLAN Notices* vol. 19 (1).
- Cohen, N. November/December 1985. Tasks as Abstraction Mechanisms. *Ada Letters* vol. 5 (3-6).
- Cohen, P., and Loisel, C. August 1988. Beyond ISA: Structures for Plausible Inference in Semantic Nets. *Proceedings of the Seventh National Conference on Artificial Intelligence*. Saint Paul, MN: American Association for Artificial Intelligence.
- Cook, W., and Palsberg, J. October 1989. A Denotational Semantics of Inheritance and Its Correctness. *SIGPLAN Notices* vol. 24 (10).
- Courtois, P., Heymans, F., and Parnas, D. October 1971. Concurrent Control with "Readers" and "Writers." *Communications of the ACM* vol. 14 (10).
- Danforth, S., and Tomlinson, C. March 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* vol. 20 (1).
- Demers, A., Donahue, J., and Skinner, G. Data Types as Values: Polymorphism, Type-Checking, Encapsulation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Dennis, J., and Van Horn, E. March 1966. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* vol. 9 (3).
- Donahue, J., and Demers, A. July 1985. Data Types Are Values. *ACM Transactions on Programming Languages and Systems* vol. 7 (3).
- Eckart, J. April 1987. Iteration and Abstract Data Types. *SIGPLAN Notices* vol. 22 (4).
- Embley, D., and Woodfield, S. 1988. Assessing the Quality of Abstract Data Types Written in Ada. *Proceedings of the 10th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- Ferber, J. October 1989. Computational Reflection in Class-Based Object-Oriented Languages. *SIGPLAN Notices* vol. 24 (10).
- Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. *IEEE Transactions on Software Engineering* vol. SE-13 (7).

- Gannon, J., McMullin, P., and Hamlet, R. July 1981. Data Abstraction Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems* vol. 3 (3).
- Gardner, M. May/June 1984. When to Use Private Types. *Ada Letters* vol. 3 (6).
- Goguen, J., Thatcher, J., and Wagner, E. 1977. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in *Current Trends in Programming Methodology: Data Structuring* vol. 4. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Graube, N. October 1989. Metaclass Compatibility. *SIGPLAN Notices* vol. 24 (10).
- Gries, D., and Prins, J. July 1985. A New Notion of Encapsulation. *SIGPLAN Notices* vol. 20 (7).
- Grogono, P., and Bennett, A. November 1989. Polymorphism and Type Checking in Object-Oriented Languages. *SIGPLAN Notices* vol. 24 (11).
- Guttag, J. 1980. Abstract Data Types and the Development of Data Structures, in *Programming Language Design*. ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Hammons, C., and Dobbs, P. May/June 1985. Coupling, Cohesion, and Package Unity in Ada. *Ada Letters* vol. 4 (6).
- Harrison, G., and Liu, D. July/August 1986. Generic Implementations Via Analogies in the Ada Programming Language. *Ada Letters* vol. 6 (4).
- Hayes, P. 1981. The Logic of Frames, in *Readings in Artificial Intelligence*. ed. B. Webber and N. Nilsson. Palo Alto, CA: Tioga.
- Hayes-Roth, F. July 1985. A Blackboard Architecture for Control. *Artificial Intelligence* vol. 26 (3).
- Hayes-Roth, F., Waterman, D., and Lenat, D. 1983. *Building Expert Systems*. Reading, MA: Addison-Wesley.
- Haynes, C., and Friedman, D. October 1987. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems* vol. 9 (4).
- Henderson, P. February 1986. Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Transactions on Software Engineering* vol. SE-12 (2).
- Herlihy, M., and Liskov, B. October 1982. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems* vol. 4 (4).
- Hesselink, W. January 1988. A Mathematical Approach to Nondeterminism in Data Types. *ACM Transactions on Programming Languages and Systems* vol. 10 (1).
- Hibbard, P., Hisgen, A., Rosenbers, J., Shaw, M., and Sherman, M. 1981. *Studies in Ada Style*. New York, NY: Springer-Verlag.
- Hilfinger, P. 1982. *Abstraction Mechanisms and Language Design*. Cambridge, MA: The MIT Press.
- Hoare, C. October 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM* vol. 17 (10).
- Hoare, C. 1985. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice/Hall International.
- Hogg, J., and Weiser, S. October 1987. OTM: Applying Objects to Tasks. *SIGPLAN Notices* vol. 22 (12).

- Jajodia, S., and Ng, P. 1983. On Representation of Relational Structures by Entity-Relationship Diagrams, in *Entity-Relationship Approach to Software Engineering*. ed C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Johnson, C., 1986. Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Knight, B. 1983. A Mathematical Basis for Entity Analysis, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- LaLonde, W., and Pugh, J. August 1985. Specialization, Generalization, and Inheritance: Teaching Objectives Beyond Data Structures and Data Types. *SIGPLAN Notices* vol. 20 (8).
- Leeson, J., and Spear, M. March 1987. Type-Independent Modules: The Preferred Approach to Generic ADTs in Modula-2. *SIGPLAN Notices* vol. 22 (3).
- Lenzerini, M., and Santucci, G. 1983. Cardinality Constraints in the Entity-Relationship Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Levesque, H. July 1984. Foundations of a Functional Approach to Knowledge Representation. *Artificial Intelligence* vol. 23 (2).
- Lindgreen, P. 1983. Entity Sets and Their Description, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Liskov, B. 1980. Programming with Abstract Data Types, in *Programming Language Design*. ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Liskov, B. May 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices* vol. 23 (5).
- Liskov, B., and Scheffler, R. July 1983. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems* vol. 5 (3).
- Liskov, B., and Zilles, S. 1977. An Introduction to Formal Specifications of Data Abstractions, in *Current Trends in Programming Methodology: Software Specification and Design* vol. 1. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Lucco, S. October 1987. Parallel Programming in a Virtual Object Space. *SIGPLAN Notices* vol. 22 (12).
- Maes, P. October 1987. Concepts and Experiments in Computational Reflection. *SIGPLAN Notices* vol. 22 (12).
- Mark, I. 1983. What is the Binary Relationship Approach?, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Markowitz, V., and Raz, Y. 1983. A Modified Relational Algebra and Its Use in an Entity-Relationship Environment, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Matsuoka, S., and Kawai, S. September 1988. Using Tuple Space Communication in Distributed Object-Oriented Languages. *SIGPLAN Notices* vol. 23 (11).

- McAllester, D., and Zabih, F. November 1986. Boolean Classes. *SIGPLAN Notices* vol. 21 (11).
- McCullough, P. October 1987. Transparent Forwarding: First Steps. *SIGPLAN Notices* vol. 22 (12).
- Mealy, G. 1977. Notions, In *Current Trends in Programming Methodology: Data Structuring* vol. 4. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Merlin, P., and Bochmann, G. January 1983. On the Construction of Submodule Specifications and Communication Protocols. *ACM Transactions on Programming Languages and Systems* vol. 5 (1).
- Meyer, B. 1987. *Programming as Contracting*, Report TR-EI-12/CO. Goleta, CA: Interactive Software Engineering.
- Minoura, T., and Iyengar, S. January 1989. Data and Time Abstraction Techniques for Multilevel Concurrent Systems. *IEEE Transactions on Software Engineering* vol. 15 (1).
- Murata, T. 1984 Modeling and Analysis of Concurrent Systems, in *Software Engineering*, ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.
- Mylopoulos, J., and Levesque, H. 1984. An Overview of Knowledge Representation. In *Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Nakano, R. 1983. Integrity Checking in a Logic-Oriented ER Model, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Newton, M., and Watkins, J. November/December 1988. The Combination of Logic and Objects for Knowledge Representation. *Journal of Object-Oriented Programming* vol. 1 (4).
- Nii, P. Summer 1986. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine* vol. 7 (2).
- Ohori, A., and Buneman, P. October 1989. Static Type Inference for Parametric Classes. *SIGPLAN Notices* vol. 24 (10).
- Pagan, F. 1981. *Formal Specification of Programming Languages*. Englewood Cliffs, NJ: Prentice-Hall.
- Parent, C., and Spaccapietra, S. July 1985. An Algebra for a General Entity-Relationship Model. *IEEE Transactions on Software Engineering* vol. SE-11 (7).
- Parnas, D. 1977. The Influence of Software Structure on Reliability, in *Current Trends in Programming Methodology: Software Specification and Design* vol. 1. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Parnas, D. 1980. Designing Software for Ease of Extension and Contraction, in *Tutorial on Software Design Techniques*, 3rd ed. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Parnas, D., Clements, P., and Weiss, D. 1983. Enhancing Reusability with Information Hiding. *Proceedings of the Workshop on Reusability in Programming*, Stratford, CT: ITT Programming.
- Pattee, H. 1973. *Hierarchy Theory*. New York, NY: George Braziller.

- Peckham, J., and Maryanski, F. September 1988. Semantic Data Models. *ACM Computing Surveys* vol. 20 (3).
- Pedersen, C. October 1989. Extending Ordinary Inheritance Schemes to Include Generalization. *SIGPLAN Notices* vol. 24 (10).
- Peterson, J. September 1977. Petri Nets. *Computing Surveys* vol. 9 (3).
- Reed, D. September 1978. *Naming and Synchronization in a Decentralized Computer System*. Cambridge, MA: The MIT Press.
- Robinson, L., and Levitt, K. 1977. Proof Techniques for Hierarchically Structured Programs, in *Current Trends in Programming Methodology: Program Validation* vol. 2. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Ross, D. July/August 1986. Classifying Ada Packages. *Ada Letters* vol. 6 (4).
- Ruane, L. January 1984. Abstract Data Types in Assembly Language Programming. *SIGPLAN Notices* vol. 19 (1).
- Rumbaugh, J. September 1988. Controlling Propagation of Operations Using Attributes on Relations. *SIGPLAN Notices* vol. 23 (11).
- Shankar, K. 1984. Data Design: Types, Structures, and Abstractions, in *Software Engineering*. ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.
- Shaw, M. 1984. The Impact of Modeling and Abstraction Concerns on Modern Programming Languages. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Shaw, M. October 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software* vol. 1 (4).
- Shaw, M. May 1989. Larger Scale Systems Require Higher-Level Abstractions. *SIGSOFT Engineering Notes* vol. 14 (3).
- Shaw, M., Feldman, G., Fitzgerald, R., Hilfinger, P., Kimura, I., London, R., Rosenberg, J., and Wulf, W. 1981. Validating the Utility of Abstraction Techniques, in *ALPHARD: Form and Content*. ed. M. Shaw. New York, NY: Springer-Verlag.
- Shaw, M., Wulf, W., and London, R. 1981. Abstraction and Verification in ALPHARD: Iteration and Generators, in *ALPHARD: Form and Content*. ed. M. Shaw. New York, NY: Springer-Verlag.
- Sherman, M., Hisgen, A., and Rosenberg, J. 1982. A Methodology for Programming Abstract Data Types in Ada. *Proceedings of the AdaTEC Conference on Ada*. New York, NY: Association of Computing Machinery.
- Siegel, J. April 1988. Twisty Little Passages. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1 (3). Everett, WA: Object Oriented Programming for Smalltalk Application Developers Association.
- Stefik, M., Bobrow, D., and Kahn, K. January 1986. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software* vol. 3 (1).
- Strom, R., and Yemini, S. January 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* vol. SE-12 (1).
- Stubbs, D., and Webre, N. 1985. *Data Structures with Abstract Data Types and Pascal*. Monterey, CA: Brooks/Cole.

- Swaine, M. June 1988. Programming Paradigms. *Dr. Dobbs's Journal of Software Tools*, no. 140.
- Tabourier, Y. 1983. Further Development of the Occurrences Structure Concept: The EROS Approach, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Tanenbaum, A. 1981. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall.
- Throelli, L. October 1987. Modules and Type Checking in PL/LL. *SIGPLAN Notices* vol. 22 (12).
- Tomlinson, C., and Singh, V. October 1989. Inheritance and Synchronization with Enabled-sets. *SIGPLAN Notices* vol. 24 (10).
- Toy, W. 1984. Hardware/Software Tradeoffs, in *Software Engineering* ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.
- Vegdahl, S. November 1986. Moving Structures between Smalltalk Images. *SIGPLAN Notices* vol. 21 (11).
- Wasserman, A. 1980. Introduction to Data Types, in *Programming Language Design* ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Weber, H., and Ehrig, H. July 1986. Specification of Modular Systems. *IEEE Transactions on Software Engineering* vol. SE-12 (7).
- Wegner, P. 6 June 1981. *The Ada Programming Language and Environment*. Unpublished draft.
- Wegner, P. 1987. On the Unification of Data and Program Abstraction in Ada, in *Object-Oriented Computing: Concepts* vol 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Wegner, P. 1987. The Object-Oriented Classification Paradigm, in *Research Directions in Object-Oriented Programming* ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Wegner, P., and Zdonik, S. August 1988. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Weihl, W., and Liskov, B. April 1985. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems* vol. 7 (2).
- Weller, D., and York, B. May 1984. A Relational Representation of an Abstract Type System. *IEEE Transactions on Software Engineering* vol. SE-10 (3).
- White, J. July 1983. On the Multiple Implementation of Abstract Data Types within a Computation. *IEEE Transactions on Software Engineering* vol. SE-9 (4).
- Wirth, N. December 1974. On the Composition of Well-structured Programs. *Computing Surveys* vol. 6 (4).
- Wirth, N. January 1983. Program Development by Stepwise Refinement. *Communications of the ACM* vol. 26 (1).
- Wirth, N. April 1988. Type Extensions. *ACM Transactions on Programming Languages and Systems* vol. 10 (2).
- Wolf, A., Clarke, L., and Wileden, J. April 1988. A Model of Visibility Control. *IEEE Transactions on Software Engineering* vol. 14 (4).
- Woods, W. October 1983. What's Important About Knowledge Representation? *IEEE Computer* vol. 16 (10).

- Zilles, S. 1984. Types, Algebras, and Modelling, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Zippel, R. June 1983. Capsules. *SIGPLAN Notices* vol. 18 (6)

К. Инструментальные средства и оболочки

- Andrews, T., and Harris, C. 1987. *Combining Language and Database Advances in an Object-Oriented Development Environment*. Billerica, MA: Ontologic
- Corradi, A., and Leonardi, L. 1986. *An Environment Based on Parallel Objects*. Bologna, Italy: Universita' di Bologna.
- Deutsch, P., and Taft, E. June 1980. *Requirements for an Experimental Programming Environment*, Report CSL-80-10. Palo Alto, CA: Xerox Palo Alto Research Center.
- Diederich, J., and Milton, J. October 1987. An Object-Oriented Design System Shell. *SIGPLAN Notices* vol. 22 (12).
- Durant, D., Carlson, G., and Yao, P. 1987. *Programmer's Guide to Windows*. Berkeley, CA: Sybex.
- Erman, L., Lark, J., and Hayes-Roth, F. December 1988. ABE: An Environment for Engineering Intelligent Systems. *IEEE Transactions on Software Engineering* vol. 14 (12).
- Ferrel, P., and Meyer, R. October 1989. Vamp: The Aldus Application Framework. *SIGPLAN Notices* vol. 24 (10).
- Fischer, H., and Martin, D. 1987. *Integrating Ada Design Graphics into the Ada Software Development Process*. Encino, CA: Mark V Business Systems.
- Goldberg, A. 1984a. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.
- Goldberg, A. 1984b. The Influence of an Object-Oriented Language on the Programming Environment, in *Interactive Programming Environments*, ed. B. Barstow. New York, NY: McGraw-Hill.
- Goldstein, I., and Bobrow, D. March 1981. *An Experimental Description-Based Programming Environment*, Report CSL-81-3. Palo Alto, CA: Xerox Palo Alto Research Center.
- Gorlen, K. May 1986. *Object-Oriented Program Support*. Bethesda, MD: National Institute of Health.
- Hecht, A., and Simmons, A. 1986. Integrating Automated Structured Analysis and Design with Ada Programming Support Environments. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Hedin, G., and Magnusson B. August 1988. The Mjolner Environment: Direct Interaction with Abstractions. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Hudson, S., and King, R. June 1988. The Cactic Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering* vol. 14 (6).

- International Business Machines. April 1988. *Operating System/2 Seminar Proceedings, IBM OS/2 Standard Edition Version 1.1, IBM Operating System/2 Update, Presentation Manager*. Boca Raton, FL.
- Kant, E. 26 March 1987. *Interactive Problem Solving with a Task Configuration and Control System*. Ridgefield, CT: Schlumberger-Doll Research.
- Kleyn, M., and Gingrich, P. September 1988. GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views. *SIGPLAN Notices* vol. 23 (11).
- Laff, M., and Hailpern, B. July 1985. SW-2 – An Object-Based Programming Environment. *SIGPLAN Notices* vol. 20 (7).
- MacLenna, B. July 1985. A Simple Software Environment Based on Objects and Relations. *SIGPLAN Notices* vol. 20 (7).
- Marques, J., and Guedes, P. October 1989. Extending the Operating System to Support an Object-Oriented Environment. *SIGPLAN Notices* vol. 24 (10).
- Minsky, N., and Rozenstein, D. February 1988. A Software Development Environment for Law-Governed Systems. *SIGPLAN Notices* vol. 24 (2).
- Moreau, D., and Dominick, W. 1987. *Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics*. Lafayette, LA: University of Southwestern Louisiana, Center for Advanced Computer Studies.
- Nakata, S., and Yamazaki, G. 1983. ISMOS: A System Based on the E-R Model and its Application to Database-Oriented Tool Generation, in *Entity-Relationship Approach to Software Engineering*, ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Nye, A. 1989. *Xlib Programming Manual for Version 11*. Newton, MA: O'Reilly and Associates.
- O'Brien, P., Halbert, D., and Kilian, M. October 1987. The Trellis Programming Environment. *SIGPLAN Notices* vol. 22 (12).
- Open Look Graphical User Interface Functional Specification*. 1990. Reading, MA: Addison-Wesley.
- OSF/Motif Style Guide, Version 1.0*. 1989. Cambridge, MA: Open Software Foundation.
- Penedo, M., Ploedereder, E., and Thomas, I. February 1988. Object Management Issues for Software Engineering Environments. *SIGPLAN Notices* vol. 24 (2).
- Reenskaug, T., and Skaar, A. October 1989. An Environment for Literate Smalltalk Programming. *SIGPLAN Notices* vol. 24 (10).
- Rosenblatt, W., Wileden, J., and Wolf, A. October 1989. OROS: Toward a Type Model for Software Development Environments. *SIGPLAN Notices* vol. 24 (10).
- Russo, V., and Campbell, R. October 1989. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 24 (10).
- Scheifler, R., and Gettys, J. 1986. The X Window System. *ACM Transactions on Graphics* vol. 63.
- Schwan, K., and Matthews, J. July 1986. Graphical Views of Parallel Programs. *Software Engineering Notes*, vol. 11 (3).
- Shear, D. 8 December 1988. CASE Shows Promise but Confusion Still Exists. *EDN* vol. 33 (25).

- Sun Microsystems. 29 March 1987. *NeWS Technical Overview* Mountain View, CA.
- Tarumi, H., Agusa, K., and Ohno, Y. 1988. A Programming Environment Supporting Reuse of Object-Oriented Software. *Proceedings of the 10th International Conference on Software Engineering*, New York, NY: Computer Society Press of the IEEE.
- Taylor, R., Belz, F., Clarke, L., Osterweil, L., Selby, R., Wileden, J., Wolf, A., and Young, M. February 1988. Foundations for the Arcadia Environment. *SIGPLAN Notices* vol. 24 (2).
- Tesler, L. August 1981. The Smalltalk Environment. *Byte* vol. 6 (8).
- Vines, D., and King, T. 1988. *Gala. An Object-Oriented Framework for an Ada Environment*. Minneapolis, MN: Honeywell.
- Vines, P., Vines, D., and King, T. 1988. *Configuration and Change Control in Gala*. Minneapolis, MN: Honeywell.
- Weinand, A., Gamma, E., and Marty, R. 1989. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming* vol. 10 (2).
- Wiorkowski, G., and Kull, D. 1988 *DB2 Design and Development Guide*. Reading, MA: Addison-Wesley.

Англо-русский толковый словарь терминов по объектно-ориентированному подходу

abstract class (абстрактный класс) — класс объектов, для которого не определены экземпляры объектов. Абстрактные классы вводятся для последующего дополнения в части структуры и поведения. Методическая часть абстрактного класса обычно является неполной и требует уточнения в производных от данного класса подклассах.

abstraction (абстракция) — существенные характеристики объекта, которые отличают его от всех других видов объектов и четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

active object (активный объект) — объект, имеющий собственные каналы управления.

actor (воздействующий объект) — объект, способный воздействовать на другие объекты и не подверженный действию со стороны других объектов. В определенном смысле этот термин совпадает с понятием «активный объект».

agent (посредник) — объект, обладающий свойством управления другими объектами и одновременно допускающий управление со стороны действующих на него объектов. Такие объекты создаются для выполнения каких-либо действий под влиянием воздействующего объекта или другого объекта посредника.

aggregate class (агрегатированный класс) — класс объектов, созданный преимущественно на основе других классов путем наследования их признаков, и в меньшей степени за счет определения собственных структур и методов.

algorithmic decomposition (алгоритмическая декомпозиция) — процесс разделения системы на отдельные фрагменты, каждый из которых отражает достаточно ограниченный этап общего сложного процесса функционирования.

base class (базовый класс) — наиболее общий класс объектов в какой-либо структуре классификации. В большинстве практических приложений определяется множество таких базовых классов. В некоторых языках программирования определяется исходный (примитивный) базовый класс, который служит в качестве первичного суперкласса для любых других классов.

behavior (поведение) — описание объекта в терминах изменения его состояния и передачи сообщений в процессе воздействия или под действием других объектов.

blocking object (блокированный объект) — пассивный объект, имеющий каналы управления, которые могут быть использованы для последующей активизации объекта.

cardinality (мощность множества объектов) — допустимое число объектов, которые могут быть определены для некоторого класса, т.е. число экземпляров для данного класса объектов, для которых могут быть установлены взаимоотношения.

class (класс) — множество объектов, связанных общностью структуры и поведения. Термины «класс» и «тип» в большинстве случаев (но не всегда) взаимозаменяемы. Понятие класса имеет некоторое принципиальное отличие от понятия типа (в смысле значимости наследственных и иерархических связей).

class category (категория классов) — группа классов, для которой определяют атрибуты видности или защиты по отношению к другим классам объектов.

class diagram (диаграмма, схема классов) — одно из изобразительных средств объектно-ориентированного проектирования, имеющее своей целью наглядно показать используемые классы и их взаимоотношение в процессе логического проектирования системы.

class structure (структура классов) — способ отражения иерархического строения системы. Это граф, вершины которого соответствуют классам, а другие — взаимоотношениям классов. Структура класса для конкретной системы представляется в ядре совокупности диаграмм (схем) классов.

class utility (поддержка класса) — совокупность общедоступных процедур.

class variable (переменная, параметр класса) — фрагмент области данных, являющийся частью информации о состоянии данного класса объектов. Совокупность всех параметров данного класса образует структуру класса. Параметры класса являются единым для всех объектов, принадлежащих данному классу.

client (объект-пользователь) — объект, который использует ресурсы другого объекта (либо действуя под управлением второго, либо в зависимости от его состояния).

concurrency (параллелизм) — свойство, допускающее одновременное существование активных и пассивных объектов. Это свойство является фундаментальным для объектной методологии.

concurrent object (параллельный объект) — активный объект, связанный каналами управления.

constructor (конструктор) — процедура создания объекта и (или) инициализация его начального состояния.

container class (сборный класс) — класс, объекты которого составлены из других объектов. Сборный класс может состоять из ряда однородных объектов (объекта одного класса), либо разнородных (относящихся к разным классам). Сборные классы могут быть обобщенными или иметь параметры, обозначающие классы содержащихся объектов.

destructor (деструктор) — процедура, которая делает состояние объекта неопределенным и (или) ликвидирует сам объект.

dynamic binding (динамическая связь) — связь имени объекта с классом, которая устанавливается только в процессе динамического порождения (создания) объекта в рабочей программе.

encapsulation (ограничение доступа) — процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта как целого. Защита в ядре ограничения доступа обычно устанавливается как в отношении структуры объекта, так и в отношении его методического содержания. Понятие «ограничение доступа» обычно соответствует понятию «защита информации». Ограничение доступа является фундаментальным свойством объектного подхода.

field (поле) — обозначает фрагмент данных о состоянии объекта, является частью его структуры и используется наряду с терминами «переменная объекта», «фрагмент объекта» и «слот» (*instance variable, member object, slot*).

free subprogram (общедоступная процедура) — процедура или функция, представляющая собой сложную операцию над объектом или объектами одного или различных классов. Такой процедурой может быть любая, не входящая в методическую часть какого-либо объекта процедура или функция.

friend (общность) — возможность использования метода двумя и более объектами различных классов, связанных отношениями общности.

function (функция) — в контексте анализа требований к системе обособленный, наблюдаемый и контролируемый фрагмент поведения.

generic function (обобщенная функция) — какая-либо операция над объектом. Обобщенная функция может быть переопределена в производном подклассе, следовательно, ее реализация зависит от всей последовательности методических описаний и наследственной иерархии. Этот термин обычно соответствует термину «виртуальная функция» (virtual function).

generic class (обобщенный класс) — класс, являющийся основой для порождения других (производных) классов с помощью дополнения другими классами, объектами и (или) операциями. Прежде чем будет создан какой-либо объект обобщенного класса, осуществляется замена его параметров конкретными значениями. Обычно обобщенные классы используются в качестве сборных классов. Понятия «обобщенный класс» и «параметризованный класс» (parameterized class) идентичны.

hierarchy (иерархия) — ранжированная и (или) упорядоченная система абстракций. Двумя наиболее употребительными видами иерархии применительно к сложным системам являются классификация (иерархия по номенклатуре и по типам) и структура объекта (иерархия составных частей). Архитектура сложных систем и их функционирование также могут представляться иерархически. Иерархичность — фундаментальное свойство объектного подхода.

identity (индивидуальность) — сущность объекта, отличающая его от всех других объектов.

implementation (реализация) — внутреннее строение класса, объекта или модуля, учитывающее особенности его поведения.

information hiding (защита информации) — обеспечение защиты всех элементов объекта. В большинстве случаев соответствует понятию «ограничение доступа» (encapsulation).

inheritance (наследование) — такое соответствие между классами, когда один класс порождает структуру или поведение другого (простое наследование) или других (множественное наследование) классов. Наследование устанавливает иерархию классов номенклатурного характера (по типу), при которой подкласс (производный) наследует признаки одного или нескольких суперклассов (исходных). Подклассы обычно имеют дополненные или переопределенные структуру и поведение по отношению к исходным суперклассам.

instance (экземпляр объекта) — конкретное представление (реализация), обладающее характеристиками состояния, поведения и индивидуальности. Структура и поведение сходных по сути экземпляров объекта описываются в общем для них классе. Соответствует понятию «объект» (object).

instance variable (переменная объекта) — фрагмент данных о состоянии объекта. Совокупность переменных объекта определяют его структуру. Соппадает с понятием «поле», «фрагмент объекта» и «слот» (field, member object, slot).

instantiation (наполнение объекта) — процесс конкретизации обобщенного или параметризованного класса, в результате которого образуется класс, позволяющий порождать экземпляры объекта.

interface (интерфейс) — внешние особенности класса, объекта или модуля, придающие ему абстрактную форму и одновременно скрывающие его внутреннее устройство и поведение.

iterator (итератор) — операция, позволяющая взаимодействовать с частями объекта.

key abstraction (ключевая абстракция) — класс или объект, являющийся частью словаря предметной области.

levels of abstraction (уровни абстракции) — относительное ранжирование абстракций в структуре классов и объектов, в архитектуре модулей и функций. В иерархии по составу

более высокий уровень абстракции представляют такие абстракции, которые составлены из других абстракций. В иерархии по номенклатуре (типам) более высокий уровень иерархии соответствует наиболее общим абстракциям, а более низкий — специализированным (уточненным) абстракциям.

mechanism (механизм) — структура, в которой организовано совместное функционирование объектов для обеспечения заданных целей.

member function (функция-элемент) — операция над объектом, являющаяся частью описания класса. Все функции-элементы — операции, но не все операции — функции-элементы. Понятие «функция-элемент» совпадает в большинстве случаев с понятием «метод» (method).

member object (фрагмент объекта) — часть данных о состоянии объекта. Совпадает с понятиями «поле», «переменная объекта», «слот» (field, instance variable, slot).

message (сообщение) — операция связи между объектами. Совпадает с понятиями «метод» (method) и «действие» (operation).

metaclass (метакласс) — класс, порождающий другие классы.

method (метод) — операция над объектом, определяемая в описании класса. Не любая операция является методом. Совпадает с понятиями «сообщение» (message), «действие» (operation). В некоторых языках метод может быть переопределен в подклассах, а в других является неизменяемой частью обобщенной функции или виртуальной функции.

mixin (смешение) — введение в класс особого дополнительного метода, который обычно противоположен методу исходного класса.

modifier (модификатор) — способ изменения состояния объекта.

modularity (модульность) — свойство системы, связанное с возможностью декомпозиции ее на ряд тесно связанных модулей. Модульность — один из фундаментальных элементов объектного подхода.

module (модуль) — фрагмент кода, являющийся строительным элементом в структуре системы; программный блок, который содержит декларации, выраженные в соответствии с требованиями языка программирования и реализующие классы и объекты системы. Как правило, модуль состоит из интерфейсной части и реализации.

module architecture (модульная архитектура) — иерархия модулей, составляющих структуру системы. Это граф, вершины которого соответствуют модулям, а дуги — соотношениям модулей между собой. Модульная архитектура представляет собой совокупность модульных диаграмм.

module diagram (модульная диаграмма) — прием в объектно-ориентированном проектировании, позволяющий графически отразить соответствие классов и объектов модулям проектируемой системы.

monomorphism (мономорфизм) — принципиальное положение теории типизации, согласно которому имена (например, объявление переменных) могут присваиваться только объектам какого-либо класса.

object (объект) — конкретное представление (реализация) абстрактного класса, обладающее характеристиками состояния, поведения и индивидуальности. Соответствует понятию «экземпляр объекта» (instance).

object diagram (диаграмма объектов) — элемент объектно-ориентированного проектирования, позволяющий наглядно отразить многообразие объектов и их соотношение в процессе логического проектирования системы. Диаграмма объектов может отражать как всю, так и часть объектной структуры системы, и направлена в первую очередь на иллюстрирование смысла ключевых механизмов проекта. Отдельная диаграмма объектов является мигновым снимком течения событий или изменения конфигурации объектов.

object model (объектный подход, методология) — совокупность основополагающих принципов, лежащих в основе объектно-ориентированного проектирования. Применительно к программной продукции такими принципами являются: абстрагирование, защита информации, модульность, иерархия, типизация, параллелизм и устойчивость.

object structure (объектная структура) — иерархическая структура, построенная по принципу составных частей. Представляет собой граф, вершины которого соответствуют объектам, а дуги — взаимоотношениям объектов. Для отражения такой структуры или ее частей используются диаграммы объектов.

object-based programming (объектное программирование) — методология программирования, основанная на представлении программы в качестве совокупности объектов, каждый из которых является реализацией определенного типа. Типы в свою очередь образуют иерархию, не связанную с принципом наследования. В таких программах типы могут рассматриваться как статические элементы, а объекты имеют более динамическую природу в рамках существующих статических связей и в соответствии с принципом мономорфизма.

object-oriented analysis (объектно-ориентированный анализ) — методология анализа, при которой требования формируются на основе понятий классов и объектов, составляющих словарь предметной области.

object-oriented decomposition (объектная декомпозиция) — процесс выделения в системе фрагментов, соответствующих классам и объектам предметной области. Практическое применение методов объектно-ориентированного проектирования связано с объектной декомпозицией, при которой любая проблема должна рассматриваться как совокупность объектов, согласованно действующих в целях обеспечения заданных требований.

object-oriented design OOD (объектно-ориентированное проектирование) — методология проектирования, соединяющая процесс объектной декомпозиции и приемы представления логической, физической, статической и динамической моделей проектируемой системы. В частности, к числу таких приемов относятся диаграммы классов, объектов, модулей и процессов.

object-oriented programming OOP (объектно-ориентированное программирование) — методология реализации, основанная на представлении программы в качестве совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследования. При этом классы являются статическими элементами, а объекты имеют динамическую природу в рамках динамических средств поддержки и в соответствии с принципами полиморфизма.

operation (действие) — определенное воздействие одного объекта на другой объект с целью вызвать соответствующую реакцию. Действия, которые можно оказать на объект, определяются либо в его методической части как функции-элементы либо относятся к числу общедоступных процедур. Соответствует терминам «сообщение» (message), «метод» (method).

parameterized class (параметризованный класс) — базовый класс, служащий основой для образования других классов, путем замены в базовом классе параметров (параметризация другими классами, объектами и (или) действиями) на значения. Только после «наполнения» параметров такого класса их значениями он становится пригодным для создания (реализации) объектов. Обычно используется в качестве сборного класса и имеет такой же смысл, как термин «обобщенный класс» (generic class).

passive object (пассивный объект) — объект, не имеющий собственных каналов управления.

persistence (устойчивость) — свойство объекта существовать во времени не зависимо от процесса, породившего данный объект и (или) в пространстве (перемещение объекта в память за пределы места порождения). Это один из основных элементов объектной методологии.

polymorphism (полиморфизм) — один из принципов теории типизации, состоящий в том, что имена (например, объявление переменных) могут соответствовать различным классам объектов, входящим в один суперкласс. Следовательно, один объект, отмеченный таким именем, может по-разному реагировать на некоторое множество действий.

private (обособленный) — часть интерфейса какого-либо класса, объекта или модуля, закрытая для доступа со стороны других классов, объектов и модулей (ограничение «видимости»).

process architecture (архитектура процессов) — отражение физической структуры системы в виде иерархии процессов. Представляет собой граф, вершины которого соответствуют процессорам и устройствам, а дуги — соединениям между ними. Для отражения архитектуры процессов используется диаграмма процессов.

process diagram (диаграмма процессов) — один из приемов объектно-ориентированного проектирования, используемый для представления процессов. Отражает полностью или частично архитектуру процессов системы.

protected (защищенный) — часть интерфейса какого-либо класса, объекта или модуля, закрытая для доступа со стороны других классов, объектов и модулей, кроме тех, которые являются производными от данного класса (подклассы).

protocol (протокол) — способ взаимодействия между объектами, отражающий их статические и динамические свойства в процессе такого взаимодействия.

public (общедоступный) — часть интерфейса какого-либо класса, объекта или модуля, открытая для доступа всем другим классам, объектам и модулям.

round-trip gestalt design (возвратное проектирование) — стиль проектирования, имеющий дискретно-итеративный характер, с постепенным уточнением физической и логической картины системы. Объектно-ориентированное проектирование является возвратным, что подчеркивает взаимное влияние целого и частного в системе.

selector (определитель состояния) — операция доступа к данным о состоянии объекта, не позволяющая изменять это состояние.

sequential object (объект-транслятор) — пассивный объект, имеющий единственный канал управления.

server (исполнитель) — объект, управляемый другими объектами, но не используемый в качестве управляющего.

slot (слот) — фрагмент области данных из структуры объекта. Соответствует терминам «поле», «переменная объекта», «фрагмент объекта» (field, instance variable, member object).

state (состояние) — один из возможных вариантов условий существования объекта. Представляет собой текущий набор свойств объекта (обычно статических) и количественных значений этих свойств (динамических).

state space (пространство состояний) — перечислимое множество всех возможных состояний объекта. Пространство состояний программы содержит неопределенное или конечное число состояний (не обязательно желаемых или ожидаемых).

state transition diagram (диаграмма перехода состояний) — фрагмент классификации, предназначенный для определения пространства состояний, допустимых для объектов данного класса; событий, вызывающих переходы из одного состояния в другое, и последствий изменений состояния.

static binding (статическая связь) — связь имени объекта (переменной) с классом. Статическая связь устанавливается в процессе компиляции до создания объекта данного класса.

strongly typed (строгий типизированный) — свойство языка программирования, в соответствии с которым все выражения должны гарантировать согласование типов операндов и параметров.

structure (структура) — конкретное состояние объекта. Каждый объект имеет собственное представление его состояния, независимое от других объектов, хотя все объекты одного класса имеют общую организацию данных, отражающих их состояние.

structured design (структурное проектирование) — метод проектирования, основанный на алгоритмической декомпозиции.

subclass (подкласс) — класс, являющийся производным от одного или нескольких других классов, которые называются непосредственными суперклассами.

subsystem (подсистема) — совокупность модулей, часть которых открыта («видима») для других подсистем, а часть защищена.

superclass (суперкласс) — класс, из которого с помощью механизма наследования создаются другие (производные) классы (называемые непосредственными подклассами).

synchronization (синхронизация) — согласованность параллельно выполняемых действий. Действия могут быть простыми (когда используется единственный канал управления), синхронными (когда согласуются два процесса), приостановленными (один процесс синхронизируется с другим только в случае, когда второй процесс находится в состоянии ожидания), задержанными (когда ожидание одним процессом другого ограничено конкретным отрезком времени) и асинхронными (когда два процесса осуществляются независимо).

thread of control (канал управления) — отдельный процесс (действие). Началом канала управления является область системы, в которой имеет место независимое динамическое воздействие. Система может иметь произвольное число каналов управления, часть которых динамически активизируется, а затем возвращается в пассивное состояние. Системы, реализуемые на нескольких процессорах, позволяют выполнять активизацию нескольких каналов параллельно. При наличии одного процессора осуществляется имитация параллельности управления.

timing diagram (временная диаграмма) — один из приемов, используемый для графической иллюстрации динамического взаимодействия объектов применительно к диаграмме объектов.

type (тип) — определение, позволяющее ограничить область данных, занимаемых объектом, и перечень действий, выполняемых над объектом. В большинстве случаев этот термин совпадает (но не всегда) с термином «класс». Небольшим принципиальным отличием термина «класс» является его ориентация на определение типа объектов.

typing (типизация) — ограничение, предъявляемое к классу объектов, препятствующее взаимозамене различных классов или (в большинстве случаев) сильно ограничивающее возможность взаимозамены. Типизация — один из основных элементов объектной методологии.

use (использование, включение) — термин для отражения внешнего проявления абстракции. Отношение использования подразумевает возможность обмена сообщениями между объектами.

visibility (видимость) — способность одной абстракции «видеть» (иметь право доступа) другую абстракцию и, следовательно, использование внешних ресурсов этой (второй) абстракции. Абстракции являются видимыми друг для друга в том случае, если их области действия пересекаются.

virtual function (виртуальная функция) — действие над объектом, которое может переопределяться в подклассе. Следовательно, фактически выполняемое действие определяется всей последовательностью методов, объявленных для данной функции в наследованных классах. Этот термин, как правило, соответствует термину «обобщенная функция» (generic function).

Предметно-именной указатель

- Абельсон 44
Абстракция 44, 508
— алгоритмическая 32
— ключевая 140, 510
Архитектура процессов 232, 513
Аткинсон 72

Бейли 138
Беранис 44
Блэк 68
Бобров 38, 188
Борнинг 66
Бриттон 51

Варда 42
Вегнер 63, 102
Вейс 56, 127
Видимость 125, 514
— модуля 163
Виртуальная функция 104, 107, 514
Влисайдес 110

Гане 42
Грей 44

Данфорт 60
Давф 106
Де Марко 42
Действие 41, 45, 512
Дейтч 62, 429
Декомпозиция 22
— алгоритмическая 22, 25, 508
— объектная 22, 511
Делегирование 96
Деструктор 50, 83, 509
Джеймс 134
Джоисон 68, 103
Диаграмма
— временная 148, 163, 172
— класса 148, 149, 151, 157, 172, 509
— модульная 148, 163, 165, 172, 511
— объектов 148, 158, 172
— процессов 148, 172
— перехода состояний 148, 158, 172

Защита информации 50, 55, 510
Зейдевиц 89
Зелковиц 55

Иерархия 20, 26, 58, 510
— классов 149
Ингалс 50, 66, 89, 122
Индивидуальность 78, 84, 510
Интерфейс 50, 510
Исполнитель 48, 513
Использование 114, 514
Итератор 67, 510

Йордан 26

Канал управления 68, 84, 514
Карделли 102
Категория 132
— классов 151, 163
Качество абстракции 122
Класс 93
— абстрактный 230, 277, 286, 453
— базовый 99, 508
— метакласс 96, 119, 511
— суперкласс 98, 514
Клемементс 56, 127
Кокс 60
Константин 26, 123
Конструктор 50, 51, 509
Коуд 137

Лим 68
Литон 110
Лисков 50, 54, 60

Майерс 26, 123
Мейер 58, 93
Меллор 42, 136
Метод 82, 511
Механизм 142, 511
— абстракции 32, 35
Микаллеф 109
Модификатор 67, 83, 117, 141, 511
Модуль 36
Модульная архитектура 164, 511
Модульность 53, 56, 511
Мономорфизм 68, 102, 511
Мощность множества объектов 508
Муре 138

Наполнение объекта 510
Наследование 60, 98, 109, 510
— множественное 61, 98, 110
— простое 60, 98

- Нейбор 138
 Нотация 146, 147, 167
 О'Брайен 124, 141
 Обособленный 94, 154, 513
 Общедоступный 94
 — общедоступная процедура 242, 305, 509
 Общность 93, 126, 509
 Объект 77, 84
 — активный 89, 508
 — блокированный 91, 508
 — воздействующий 508
 — метаобъект 121
 — параллельный 91, 509
 — пассивный 512
 — пользователь 44, 509
 — транслятор 91, 513
 Объектно-ориентированное программирование 512
 Объектно-ориентированный анализ 42, 137, 512
 Объектный подход 20, 31, 512
 Ограничение доступа 50, 509
 Определитель состояния 50, 513
 Пайджет 133
 Параллелизм 68, 509
 Парнас 51, 56, 127
 Переменная объекта 95, 510
 Пирбой 42
 Переменная, параметр класса 67, 104, 155, 509
 Поведение 81, 508
 Поддержка класса 509
 Подкласс 98, 514
 Поле 95, 126, 509
 Полноморфизм 67, 102, 513
 — множественный 113
 — простой 101
 Посредник 89, 91
 Проектирование 27
 — возвратное 174, 513
 — объектно-ориентированное 26, 42, 512
 — структурное 22, 514
 Пространство состояний 172, 513
 Протокол 44, 513
 Рабсон 74
 Реализация 51, 510
 — внутренняя 51
 Робсон 119
 Росс 137
 Сарсон 42
 Связь
 — динамическая 66, 513
 — статическая 66, 513
 Сейдлиц 44
 Семантические сети 99, 172
 Синхронизация 91, 125, 160, 514
 Слот 53, 95, 513
 Смещение 112, 511
 Смит 77
 Снайдер 93
 Сообщение 81, 511
 Состояние 78, 513
 Стайн 25
 Старк 44
 Стефик 188
 Стивенс 123
 Страустрап 39, 53, 107, 118, 129, 141
 Структура 58, 109, 514
 — классов 58, 81, 94, 509
 — объектов 58, 512
 Суссман 44
 Теслер 66
 Тип 33, 39, 62, 514
 Типизация 62 514
 — нестрогая 63
 — строгая 63, 514
 Токей 77
 Томлисон 60
 Уинстон 108
 Устойчивость 73 512
 Фрагмент объекта 95, 511
 Функция 510
 — обобщенная 38, 121, 510
 — элемент 45, 82, 511
 Хальберт 124, 141
 Хатли 42
 Хендлер 113
 Хоаре 26
 Хорн 108
 Шклаер 136
 Шоу 44, 131
 Эббот 138
 Экземпляр объекта 47, 64, 510

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА	5
ВВЕДЕНИЕ	7
Часть I. КОНЦЕПЦИИ	10
Глава 1. СЛОЖНОСТЬ	10
1.1. СЛОЖНОСТЬ, ПРИСУЩАЯ ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ	10
1.2. СТРУКТУРА СЛОЖНЫХ СИСТЕМ	16
1.3. ВНЕСЕНИЕ ПОРЯДКА В ХАОС	22
1.4. ПРОЕКТИРОВАНИЕ СЛОЖНЫХ СИСТЕМ	27
Глава 2. ОБЪЕКТНЫЙ ПОДХОД	31
2.1. СТАНОВЛЕНИЕ ОБЪЕКТНОГО ПОДХОДА	31
2.2. КОМПОНЕНТЫ ОБЪЕКТНОГО ПОДХОДА	42
2.3. ПРИМЕНЕНИЯ ОБЪЕКТНОГО ПОДХОДА	73
Глава 3. КЛАССЫ И ОБЪЕКТЫ	77
3.1. ОБЪЕКТ	77
3.2. ОТНОШЕНИЯ МЕЖДУ ОБЪЕКТАМИ	89
3.3. СУЩНОСТЬ «КЛАСС»	93
3.4. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ	95
3.5. ВЗАИМОСВЯЗЬ КЛАССОВ И ОБЪЕКТОВ	121
3.6. ВОПРОСЫ КАЧЕСТВА ПРИ СОЗДАНИИ КЛАССОВ И ОБЪЕКТОВ	122
Глава 4. КЛАССИФИКАЦИЯ	129
4.1. ВАЖНОСТЬ ПРАВИЛЬНОЙ КЛАССИФИКАЦИИ	129
4.2. ИДЕНТИФИКАЦИЯ КЛАССОВ И ОБЪЕКТОВ	132
4.3. КЛЮЧЕВЫЕ АБСТРАКЦИИ И МЕХАНИЗМЫ	140
Часть II. МЕТОДОЛОГИЯ	146
Глава 5. СИСТЕМА ОБОЗНАЧЕНИЙ	146
5.1. ЭЛЕМЕНТЫ СИСТЕМЫ ОБОЗНАЧЕНИЙ	146
5.2. ДИАГРАММА КЛАССОВ	149
5.3. ДИАГРАММЫ ПЕРЕХОДА СОСТОЯНИЙ	157
5.4. ДИАГРАММА ОБЪЕКТОВ	158
5.5. ВРЕМЕННАЯ ДИАГРАММА	162
5.6. МОДУЛЬНАЯ ДИАГРАММА	163
5.7. ДИАГРАММЫ ПРОЦЕССОВ	168
5.8. ПРИМЕНЕНИЕ СИСТЕМЫ ОБОЗНАЧЕНИЙ	171

Глава 6. ПРОЦЕСС	173
6.1. ПРОЕКТИРОВАНИЕ КАК ПОСТУПАТЕЛЬНЫЙ ИТЕРАТИВНЫЙ ПРОЦЕСС	173
6.2. ИДЕНТИФИКАЦИЯ КЛАССОВ И ОБЪЕКТОВ	176
6.3. ИДЕНТИФИКАЦИЯ СЕМАНТИКИ КЛАССОВ И ОБЪЕКТОВ	177
6.4. ИДЕНТИФИКАЦИЯ СВЯЗЕЙ МЕЖДУ КЛАССАМИ И ОБЪЕКТАМИ	178
6.5. РЕАЛИЗАЦИЯ КЛАССОВ И ОБЪЕКТОВ	180
Глава 7. ТРАДИЦИОННЫЕ МЕТОДЫ	182
7.1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ В ЖИЗНЕННОМ ЦИКЛЕ РАЗРАБОТКИ	182
7.2. УПРАВЛЕНИЕ ПРОЕКТОМ	190
7.3. ДОСТОИНСТВА И НЕДОСТАТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ	199
Часть III. ПРИМЕНЕНИЯ	204
Глава 8. SMALLTALK. СИСТЕМА ДОМАШНЕГО ОТОПЛЕНИЯ	204
8.1. АНАЛИЗ	204
8.2. ПРОЕКТИРОВАНИЕ	221
8.3. РЕАЛИЗАЦИЯ	236
8.4. МОДИФИКАЦИЯ	252
Глава 9. ОБЪЕКТ PASCAL. ИНСТРУМЕНТАЛЬНОЕ СРЕДСТВО РАЗРАБОТКИ КОНСТРУКЦИЙ ГЕОМЕТРИЧЕСКОЙ ОПТИКИ	254
9.1. АНАЛИЗ	254
9.2. ПРОЕКТИРОВАНИЕ	278
9.3. РАЗВИТИЕ	301
9.4. МОДИФИКАЦИЯ	331
Глава 10. C++. СИСТЕМА РЕГИСТРАЦИИ ОШИБОК В ПРОГРАММНЫХ СРЕДСТВАХ	334
10.1. АНАЛИЗ	334
10.2. ПРОЕКТИРОВАНИЕ	349
10.3. РАЗВИТИЕ ПРОЕКТА	362
10.4. МОДИФИКАЦИЯ	369
Глава 11. COMMON LISP OBJECT SYSTEM. СИСТЕМА ДЕШИФРОВАНИЯ	371
11.1. АНАЛИЗ	373
11.2. ПРОЕКТИРОВАНИЕ	377
11.3. ОБЪЕДИНЕНИЕ В СИСТЕМУ	388
11.4. МОДИФИКАЦИЯ	397

Глава 12. ADA. СИСТЕМА УПРАВЛЕНИЯ ДВИЖЕНИЕМ	400
12.1. АНАЛИЗ	403
12.2. ПРОЕКТИРОВАНИЕ	408
12.3. РАЗВИТИЕ	419
12.4. ИЗМЕНЕНИЯ	423
ЗАКЛЮЧЕНИЕ	426
ПРИЛОЖЕНИЯ	427
П.1. Введение	427
П.2. ЯЗЫК SMALLTALK	428
П.3. ЯЗЫК OBJECT PASCAL	432
П.4. Язык C++	436
П.5. ЯЗЫК COMMON LISP OBJECT SYSTEM (CLOS)	440
П.6. ЯЗЫК ADA	442
П.7. ДРУГИЕ ЯЗЫКИ ООР	446
ЛИТЕРАТУРА	447
БИБЛИОГРАФИЯ	463
АНГЛО-РУССКИЙ ТОЛКОВЫЙ СЛОВАРЬ ТЕРМИНОВ ПО ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ ПОДХОДУ	508
ПРЕДМЕТНО-ИМЕННОЙ УКАЗАТЕЛЬ	515

Фирма "Диалектика" готовит к выпуску в свет следующую литературу по программированию :

- Turbo Vision для языка Pascal
- Справочное руководство по FoxPro 2.0 в трех томах
(совместное издание с АО "И.В.К. ")
- Программирование на C++ под Windows
(совместное издание с АО "И.В.К. ")
- Turbo Pascal для Windows
- Работа с библиотекой ObjectWindows

ВНИМАНИЕ ! указанные книги будут издаваться малыми тиражами.

Желающим приобрести данную литературу необходимо выслать заявку с указанием названия книги и количества экземпляров (не забудьте указать обратный адрес). Фирма "Диалектика" просит своих подписчиков сообщать интересующие Вас темы, а также указывать языки программирования и инструментальные средства, которые Вы используете.

По мере издания книг, приславшие заявки будут уведомлены о способе приобретения литературы.

Оптовым покупателям предоставляется скидка.
Мы готовы рассмотреть любые ваши предложения.
Звоните по тел. (044) 517-88-81 в г. Киеве.

Заявки присылать по адресу :

Украина, г.Киев ул.Пархоменко, 14 салон КОМТЕХ

с грифом "Библиотека программиста"

Научно-проектная фирма

"ДИАЛЕКТИКА"

совместно с АО "И. В. К."

предлагает книги по новейшим технологиям
программирования, которые будут изданы в IV квартале
1992 года:

Книга Питера Нортона и Поля Иао

**" Программирование на BORLAND C ++
для WINDOWS "** 536 стр., твердый переплет

телефоны :

- Фирма "Диалектика" (044) 517-8881 в г.Киеве

- "И.В.К. - Софт" (095) 284-3363 в г.Москве

АННОТАЦИЯ :

Одной из сложнейших задач, появляющихся при разработке прикладных программ для WINDOWS, является упорядочение громадного числа обращений к интерфейсу прикладных программ среды WINDOWS (API). Эта задача значительно упрощается при использовании разработанной фирмой BORLAND библиотеки ObjectWindows (OWL), однако для создания "послушных" прикладных программ, полностью соответствующих стандартам WINDOWS, необходимо знать как эти программы сочетаются, и как происходит вызов различных функций WINDOWS.

Книга предназначена для программистов, чем объясняется большое количество листингов программ. Однако эта книга не просто набор листингов программ, так как в ней также подробно рассматриваются концепции и философия WINDOWS. Borland C++ дает Вам средства управления WINDOWS; книга же даст Вам навыки эффективного использования этих средств. Рассматривая библиотеку ObjectWindows, авторы наглядно демонстрируют, как быстро и эффективно можно создавать прикладные программы с использованием всех предоставляемых WINDOWS средств - текста, графики, меню, диалоговых окон и других.

Несмотря на то, что большинство прикладных программ для WINDOWS сейчас пишутся на C, можно с уверенностью предсказать, что C++ скоро займет доминирующее положение. C++ дает возможность добавления объектного программирования к уже написанным на C системам. Именно по этой причине была создана версия книги для C++.

Эта книга написана для программистов, работающих с языком C++, с целью обучения программированию в среде WINDOWS. В книге приводятся тексты программ, дающие возможность более полного ознакомления с используемой системой и ее возможностями.

Книга состоит из шести частей. В первой части приводится краткая история WINDOWS , а также описываются три новых дополнения, с которыми сталкивается программист: создание программ, управляемых сообщениями, создание и управление графическим выводом, использование объектов пользовательского интерфейса.

Во второй части рассматриваются проблемы, связанные с созданием минимальной для WINDOWS программы, а так же некоторые базовые положения программирования в среде WINDOWS . В этой части также рассматривается структура, общая для всех программ в WINDOWS.

В третьей части описывается интерфейс графических устройств (GDI). Этот интерфейс используется для создания аппаратно-независимого графического вывода.

В четвертой части рассматриваются три ключевых объекта пользовательского интерфейса: меню, окна, диалоговые окна. В этой части описываются внутренние свойства каждого из объектов и основные способы их использования.

Пятая часть посвящена средствам ввода. В этой части приводится подробное описание процесса ввода данных с физического устройства через системные буфера в программу.

В шестой части рассматриваются особенности операционной системы. Эта часть включает два больших раздела: линкование в памяти и динамическое линкование. Одной из отличительной особенности версии WINDOWS 3.0 являются изменения при использовании памяти. Для того, чтобы понять понять эти изменения, приводится подробное описание каждого из операционных режимов WINDOWS.

Эта книга станет важной частью Ваших знаний о программировании в среде WINDOWS.

Принимаются заявки по адресу : Украина, г. Киев ул.Пархоменко, 14 салон КОМТЕХ с грифом "Библиотека программиста". Не забудьте вложить почтовую открытку с указанием названия книги, количества экземпляров и обратным адресом.

"Диалектика" & "И.В.К.- Софт"
Современная литература для программистов

ДЛЯ ЗАМЕТОК

"Диалектика" & "И.В.К.- Софт"
Современная литература для программистов

ДЛЯ ЗАМЕТОК

"Диалектика" & "И.В.К.- Софт"
Современная литература для программистов

ДЛЯ ЗАМЕТОК

"Диалектика" & "И.В.К.- Софт"
Современная литература для программистов

ДЛЯ ЗАМЕТОК

"Диалектика" & "И.В.К.- Софт"
Современная литература для программистов

ДЛЯ ЗАМЕТОК



